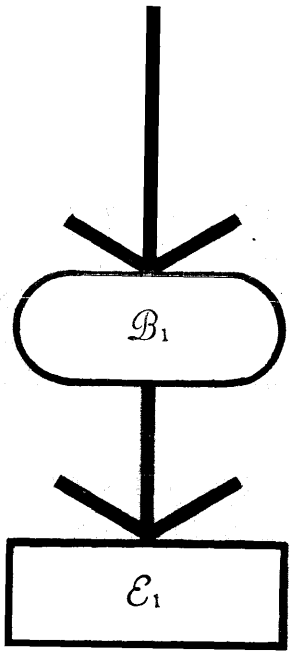


BAC-220

BURROUGHS ALGEBRAIC COMPILER

REVISED EDITION ******

220-21017
MARCH 1963

BAC-220

BURROUGHS ALGEBRAIC COMPILER

REVISED EDITION

Sales Technical Services
EQUIPMENT & SYSTEMS
MARKETING DIVISION

Burroughs Corporation
Detroit 32, Michigan

Previous Edition

Revised Edition March 1963

preface

This document supersedes Bulletin 220-21011-D, BURROUGHS *Algebraic Compiler*, dated January 1961.

Since its first installation in March of 1960, the BURROUGHS Algebraic Compiler for the 220 Electronic Data Processing System has undergone changes and additions aimed at improving the operation of the compiler, and removing restrictions on its use.

This manual describes the BURROUGHS Algebraic Compiler for the 220 in its present form; tapes containing this version are dated February 1962. Beyond the usual corrections associated with normal compiler maintenance, no further compiler changes are planned.

The February 1962 compiler system, described in this Reference Manual, includes a generator with the ability to produce a compiler conforming to *any* BURROUGHS 220 system installation.

Input-output techniques have been included which enable the programmer to write his own input-output procedures and to integrate them easily with the compiler. As a result of this change, the input and output devices employed at object time need not be the same as those employed at compile time.

The diagnostic facilities have been improved in two ways:

More extensive diagnosis is now possible, because of the addition of new error messages.

Diagnostic aids have been reorganized into categories which are easier to use and to understand.

The writing and inclusion of external procedures have been facilitated by the addition of several new pseudo-operations.

table of contents

I. INTRODUCTION 1-1

II. ELEMENTS OF THE COMPILER LANGUAGE 2-1
- Characters 2-1
- Identifiers 2-1
- Quantities 2-2
- Variables 2-2
- Constants 2-2
- Evaluated Functions 2-3

III. EXPRESSIONS 3-1
- Arithmetic Expressions 3-1
- Relations 3-2
- Boolean Expressions 3-3

IV. STATEMENTS 4-1
- The Assignment Statement 4-1
- The Grammar of Statements 4-2

V. BASIC DECLARATIONS 5-1
- Declarations of Type 5-1
- The ARRAY Declaration 5-2
- The COMMENT Declaration 5-3
- The FINISH Declaration 5-3

VI. BASIC CONTROL STATEMENTS 6-1
- Transfer of Control 6-1
- Suspension of Computation 6-2
- Clauses 6-2
- Conditional Execution 6-2
- Control of Iterations 6-4

VII. SUBPROGRAMS 7-1
- Subroutines 7-1
- Functions 7-2
- Intrinsic Functions 7-3
- Procedures 7-4
- External Program Declarations 7-7

VIII. INPUT-OUTPUT TECHNIQUES 8-1
- Input of Information 8-1
- Output of Information 8-3
- Construction of Formats 8-4

IX. OVERLAY TECHNIQUES 9-1
- The SEGMENT Declaration 9-1
- The OVERLAY Statement 9-2

X. DIAGNOSTIC FACILITIES 10-1
- Aids Available at Compile Time 10-1
- Diagnostic Aids to Object Program Execution 10-4
- Aids Specified by the Programmer 10-4
- Error Messages from Library Procedures 10-6
- Object Program Listing 10-6

XI. PROGRAMS IN ALGOL 11-1
- Examples of Programs 11-1

APPENDIX A. OPERATING INSTRUCTIONS FOR THE BAC-220 GENERATOR PROGRAM A-1
- Readying the Equipment A-1
- Running the Generator Program A-1
- Composition of Generator Input Deck A-1
- Standard Version of BAC-220 A-2
- Non-Standard Versions of BAC-220 A-5
- Special Input-Output Routines A-8
- Error Messages A-9

APPENDIX B. COMPILER OPERATING INSTRUCTIONS B-1
- Preparation of Source Programs on Punched Cards B-1
- Preparation of Paper-Tape Source Programs B-1
- Compiling a Program B-2
- Operation of the FINISH Declaration B-2
- Reloading the Object Program from Magnetic Tape B-3
- Dumping a Compiled Object Program on Cards B-3
- Reloading the Object Program from Cards B-3
- Dumping a Compiled Program on Paper Tape B-3

APPENDIX C. LIST OF RESERVED IDENTIFIERS C-1

APPENDIX D. SYNTACTICAL DESCRIPTION OF THE COMPILER LANGUAGE D-1
- Forms of Definition D-1
- Basic Symbols D-1
- Expressions D-2
- Statements D-3

APPENDIX E. TRANSLITERATION RULES E-1
- Basic Symbols E-1
- Expressions E-3
- Statements E-3
- Declarations E-4

Table of Contents (continued)

APPENDIX F. CONSTRUCTION OF MACHINE-LANGUAGE PROGRAMS F-1
Linkage to Procedures F-1
Parameters of Procedures F-1
Relocation Conventions F-2
Magnetic-Tape Operations F-3
Use of Equivalence Cards F-3
Description of Name Cards F-4
Preparation of External Programs F-5
Library Procedures F-7
The Error-Message Procedure F-7
Input-Output Procedures F-8
The FORMAT Declaration F-11

APPENDIX G. LIBRARY PROCEDURES G-1
Specimen Description G-2
FLOAT G-3
FIX G-4
WRITE G-5
READ G-6
SQRT G-7
EXP G-8
LOG G-9
FLFL G-10
FLFX G-11
FXFL G-12
FXFX G-13
SIN G-14
COS G-15
TAN G-16
ARCSIN G-17
ARCCOS G-18

APPENDIX G (continued)
ARCTAN G-19
SINH G-20
COSH G-21
TANH G-22
ROMXX G-23
ENTIRE G-24

INDEX I-1

TABLES AND ILLUSTRATIONS

TABLE 1 Some Input-Output Combinations A-2

FIGURE 1 Generator Input Deck to Create Standard Version (Combination 1) A-3
2 Statements Required for Standard Library Deck A-5
3 Generator Input Deck to Create HIGH-SPEED PRINTER Version (Combination 2) A-7
4 Generator Input Deck to Create Paper-Tape Version (Combination 3) A-8
5 Generator Input Deck to Create Combination 4 A-10
6 Generator Input Deck to Create Combination 5 A-11
7 Generator Input Deck to Create Combination 6 A-12
8 Generator Input Deck to Create Combination 7 A-13

SYSTEM REQUIREMENTS

ORGANIZATION OF THE MANUAL

ADDITIONAL COPIES

I . . .

introduction

THIS BOOK IS INTENDED as a reference manual in the use of BAC-220—the BURROUGHS Algebraic Compiler for the 220 Electronic Data Processing System. BAC-220 is a representation of ALGOL, the international algorithmic language; it accepts symbolic programs and produces machine-language programs for the BURROUGHS 220. A full description of ALGOL and its history is available elsewhere.†

Part of the BAC-220 system is a generator program, which generates different versions of the compiler in order to accommodate the diverse operating procedures and hardware configurations found in different B 220 installations. Thus the BAC-220 generator program produces versions of the compiler which preserve portions of core or tape storage, or which employ magnetic tape, punched cards, punched paper tape, the HIGH-SPEED PRINTER or the SUPERVISORY PRINTER in various combinations for input and output purposes. Directions for using the BAC-220 generator program are given in APPENDIX A.

The standard version of BAC-220 produced by the generator program requires a B 220 system comprising 5,000 words of core storage, two magnetic-tape storage units, and CARDATRON with one input and one output (LINE PRINTER) station. This standard version of BAC-220 consists of approximately 3,600 instructions and 2,000 words of stored tables.

BAC-220 accepts symbolic programs formulated according to the rules in this manual, and assembles machine-language coding to perform the indicated operations. The assembly of this coding proceeds at approximately 500 words per minute. Appropriate library and diagnostic routines, as well as a loader, are automatically included with the compiled program as part of the compilation process. Upon completion of the compilation process the compiled program may be loaded and executed immediately, or its execution may be deferred by retaining the compiled program on magnetic tape, or by having the program punched onto cards or paper tape.

As a representation of ALGOL, BAC-220 differs in certain respects from the ALGOL reference language. For example, the ALGOL character set includes lower-case letters of the alphabet, whereas representations of ALGOL—such as BAC-220—written for use on most conventional data-processing equipment recognize only upper-case letters. Also, BAC-220 provides additions for the ALGOL reference language which are essential to the operation of data-processing systems: input-output facilities, conventions for inclusion of segments of machine-language coding, and diagnostic features. Again, the order in which certain constructs (e.g., procedure declarations and call statements) appear in an ALGOL program is not specified in the definition of the reference language, whereas BAC-220, and other ALGOL representations, contain such restrictions in some cases, in order to maximize compiler efficiency.

A syntactical description of BAC-220 is presented in APPENDIX D, and a summary of transliteration rules for equivalent constructs of ALGOL and BAC-220 is provided in APPENDIX E.

The text of this manual consists of definitions, directions, and rules for the use of the compiler, examples of these definitions and rules, and some sample programs. Appendices are included to deal with specific details of the compiler and its operation. Whenever a term is defined, it is italicized in the defining sentence. References in the index to such definitions are also italicized. Script letters are used in the text to denote generic representations; e.g., $\mathcal{E}$ is used to represent an expression and $\mathcal{S}$ to represent a statement.

† See *Communications of the ACM*, vol. 1, no. 12, pp. 8-22; and vol. 3, no. 5, pp. 299-313.

BURROUGHS CORPORATION is pleased to work with the Cooperating Users of Burroughs Equipment in the task of maintaining this and other literature concerning the BURROUGHS 220.

Constructive criticisms of the contents of this manual will be appreciated by the authors and publishers. Please address them to:

MANAGER, AUTOMATIC PROGRAMING
BURROUGHS CORPORATION
460 Sierra Madre Villa
Pasadena, California, USA

Additional copies of this manual may be obtained from your BURROUGHS CORPORATION representative.

CHARACTER SETS
IDENTIFIERS
QUANTITIES
VARIABLES
CONSTANTS

II . . .

elements of the compiler language

CHARACTERS

The Burroughs Algebraic Compiler employs a character set which is commonly available as a variant of the usual Hollerith code.† These characters are:

Scientific Character Set	Hollerith Equivalents
The Roman Alphabet	
A, B, ..., Z	A, B, ..., Z
The Arabic Numerals	
0, 1, ..., 9	0, 1, ..., 9
Special Characters	
+	&
−	− or @
=	#
(	%
)	□
.	.
,	,
;	$
/	/
*	*
⟨space⟩	⟨space⟩

In addition, some multiples of characters are given meaning as though they constituted a single character:

..	:
...	ellipsis
**	base-10 scale factor appended

† All of the examples in this manual are printed with the scientific character set. If card equipment with Fortran characters is used, these same characters will be printed, with the exception of the semicolon (;), which will print as a dollar sign ($). (The type wheel is variation 'F'.) Card equipment with 'standard' characters will print in Hollerith equivalents.

From these characters statements are constructed which are translated by the compiler into a form suitable for execution by the Burroughs 220.

Metalinguistic Symbols

In addition to the script letters used in the text, some symbols will be employed with metalinguistic significance. These symbols include:

SYMBOL	SIGNIFICANCE
~ ::=	is equivalent to / has the form of
⋯	ellipsis
⟨ ⟩	broken brackets
ρ	relational operator
o	arithmetic operator
|	or
#	space

IDENTIFIERS

The fundamental construct of the compiler language is the identifier. Identifiers are used to name the various things which make up a program, for example, variables, functions, labels, and subroutines. An *identifier* is composed of a string of letters, or letters and digits, not exceeding 50 characters in length.‡ The first character of an identifier must be a letter; no special characters (including spaces) may be imbedded within an identifier.

In addition, a few identifiers are reserved for special use as operators, punctuation marks, and as names of library functions.

‡ See Chapter X for restriction on the length of identifiers to be monitored.

These reserved identifiers may not be used by the programmer in any context other than that set down in this manual. A list of the reserved identifiers is given in APPENDIX C. Any other identifiers may be used at will.

EXAMPLES:

```
Z
GAMMA
SN2N1
SIERRAMADREVILLAAVENUE
A374
YOUNGLADYOFCHICHESTER540
RUNGEKUTTAGILL
```

QUANTITIES

The BURROUGHS Algebraic Compiler is concerned with the manipulation of three kinds of quantities: *floating-point quantities*, *integer quantities*, and *Boolean quantities*. The terms FLOATING (or REAL), INTEGER, and BOOLEAN define the *type* of quantity.

Floating-point quantities are numbers which may have an integral and a fractional part. They are used to represent the class of real numbers to a precision of eight significant digits, the maximum permitted by the floating-point word format of the BURROUGHS 220. The magnitude of a floating-point quantity must be less than 10^{50}. Any floating-point quantity which is smaller in magnitude than 0.1×10^{-50} is treated as zero.

Integer quantities are those numbers which do not have a fractional part, and which represent the class of integers which can be expressed in the word length of the BURROUGHS 220, i.e., integer quantities must be smaller than 10^{10} in magnitude.

Boolean quantities represent truth values. The only values for Boolean quantities are the integer one, meaning ***true***, and zero, meaning ***false***.

A program may contain quantities of any or all of these three types. The programmer assigns the types of the variables, evaluated functions, and expressions which appear in his program. (See CHAPTER V.) The type of a constant depends upon context and form.

VARIABLES

Variables treated by this compiler are of two kinds—simple variables and subscripted variables. A *simple variable* represents a single quantity and is denoted by an identifier; a *subscripted variable* represents a single element of an array and is denoted by the identifier which names the array, followed by a subscript list enclosed in parentheses. The list consists of arithmetic expressions separated by commas.

EXAMPLES:

Simple Variables

```
X
ALPHA
C13
```

Subscripted Variables

```
A(I,J)
M(I + 1, J + 1)
V(F(P + 1), 12 + Q)
Z(W(T), X(T), Y(T), Z(T))
C(13)
```

The expressions (see CHAPTER III) which make up a subscripted variable may be of any complexity. Even floating-point values are allowed, in which case the floating-point number is truncated—the fractional part dropped—to an integer. Each subscript must have a value which is not less than unity and not greater than the maximum specified for that array by the ARRAY declaration (see CHAPTER V). The number of subscript expressions must equal the number of dimensions of the array.

Whether a variable represents a floating-point, integer, or Boolean quantity is determined by the 'declaration of type' described in CHAPTER V.

CONSTANTS

Integer Constants

Integer constants are represented by a string of digits. A maximum of ten significant digits is allowed, i.e., leading zeros are ignored. Spaces may not be imbedded within an integer constant.

EXAMPLES:

```
    0
   17
16384
 2111
```

Floating-Point Constants

Floating-point constants are represented by a string of digits which contains '.'—a decimal point. *The decimal point may not appear at the beginning or end of the string; it must be imbedded within it.* A floating-point constant may contain a maximum of eight significant digits. Leading zeros are not counted toward this maximum.

EXAMPLES:

```
 0.0
 3.1415927
43.0
 0.00006174205
```

If desired, a scale factor may be appended to a floating-point constant to indicate that it is to be multiplied by the indicated power of 10. This scale factor is written as two asterisks followed perhaps by a '+' or '−' sign and then by an integer. The integer specifies the power of 10 to be used, and is limited to a two-digit number. The magnitude of such a floating-point number must not exceed $0.99999999 \times 10^{49}$.

EXAMPLES:

2.6**5 means 2.6×10^{5} or 260,000
1.7**−3 means 1.7×10^{-3} or 0.0017

A third option allows a floating-point number to be written as an integer followed by a scale factor.

EXAMPLE:

3**+4

This is precisely equivalent to writing 3.0**+4 or 30,000.0. Note that a scale factor alone may not be used to specify a floating-point number—it is not valid to use **−2 to indicate 10^{-2}; it must be written 1**−2 or 1.0**−2, etc.

Boolean Constants

Only two *Boolean constants* are allowed: Zero (written as 0) means ***false***, and one (written as 1) means ***true***.

EVALUATED FUNCTIONS

The compiler allows the use of a wide variety of functions. In this section we will consider only the simplest form of functional notation in order to provide a basis for the next chapter. (CHAPTER VII contains a complete description of the use of functions and the manner in which they are defined.) For the moment we will assume that a function acts on one or more quantities called *arguments* and produces a single quantity as a result. This resulting quantity is called an *evaluated function.*

GENERAL FORM:

$\mathcal{g}(\mathcal{E}_1, \ldots, \mathcal{E}_q)$

where $\mathcal{g}$ is an identifier which names the function and $\mathcal{E}_1$ through $\mathcal{E}_q$ are expressions which are the arguments of the function.

EXAMPLES:

```
SIN(X)
SQRT(B*2−4.A.C)
HYPERGEOM(A,B,C,Z)
LOG(SIN(THETA−ALPHA/2))
PEIRCE(P,Q)
```

The type of a function depends on the manner in which the function was defined. The type required for each of the arguments is also determined by the definition of the function. *It is the programmer's responsibility to ensure that each of the arguments is of the proper type.*

ARITHMETIC EXPRESSIONS
BOOLEAN EXPRESSIONS
RELATIONS

III . . .

expressions

THE COMPILER deals with two kinds of expressions: Arithmetic expressions (those having numerical values) and Boolean expressions (those having truth values). This chapter describes the manner in which these expressions may be combined to produce new expressions. Expressions must be well formed in accordance with mathematical convention and with the rules set forth below.

ARITHMETIC EXPRESSIONS

Arithmetic quantities are combined by means of the operators + − · / and *. The symbol * is used to denote exponentiation, that is, B*2 has the meaning B^2. In addition to these five symbols, parentheses are employed to indicate that a specific order of evaluation is to be followed rather than the conventional order of evaluation. To be explicit, it is assumed—in the absence of parentheses to indicate otherwise—that exponentiation is performed before multiplication, multiplication before division, and division before addition and before subtraction. Convention in writing algebraic expressions suggests this ordering rather than that of assigning equal precedence to multiplication (·)† and division (/). As is customary in mathematical literature, the expressions A/B/C and A*B*C are regarded as ambiguous. Parentheses should be used to express the exact meaning desired.

A variable, a constant, or an evaluated function of floating or integer type will in itself constitute an arithmetic expression. Furthermore, if $\mathcal{E}_1$ and $\mathcal{E}_2$ are any arithmetic expressions and $\mathcal{E}_3$ is an arithmetic expression the first character of which is not a + or −, then each of the following combinations is also an arithmetic expression:

$\mathcal{E}_1 \cdot \mathcal{E}_2$	$\mathcal{E}_1 + \mathcal{E}_3$
$\mathcal{E}_1 / \mathcal{E}_2$	$\mathcal{E}_1 - \mathcal{E}_3$
$\mathcal{E}_1 * \mathcal{E}_2$	$+\mathcal{E}_3$
$(\mathcal{E}_1)$	$-\mathcal{E}_3$

† Represented on card equipment as a decimal point (.).

If $\mathcal{E}_1$ and $\mathcal{E}_2$ are both constants, the programmer must write $(\mathcal{E}_1) \cdot (\mathcal{E}_2)$, to avoid conflict with the notation for constants.

EXAMPLES:

```
X + Y*2
(720.0)12
C.SIN(N.3.1415927.F)
ARCTAN(HORIZ/VERTL) - ALPHA
(-B + SQRT(B*2 - 4.A.C))/2.A
-(Z13*-3 + Z14*-3)/17.2
A(1 + J, J + 1) - A(I + 1,1)/V(J)
```

Omission of the Multiplication Operator

In certain instances the '·' representing multiplication may be omitted. In general, this omission is possible wherever the lack of a '·' will not result in ambiguity.

More specifically, suppose that:

$\mathcal{g}$ is an identifier specifying a simple variable, an array, or a function;
$\mathcal{V}$ is an identifier specifying a simple variable;
$\mathfrak{N}$ is any constant; and
$\sim$ is the symbol for *is equivalent to*; then—

$)\mathcal{g} \sim)\cdot\mathcal{g}$
$\mathcal{V}(\sim \mathcal{V}\cdot($
$)\mathfrak{N} \sim)\cdot\mathfrak{N}$
$\mathfrak{N}(\sim \mathfrak{N}\cdot($
$) (\sim)\cdot($
$\mathfrak{N}\mathcal{g} \sim \mathfrak{N}\cdot\mathcal{g}$

EXAMPLES:

```
4A.C
3(A + B)(A - B)
TAN(2X)ALPHA
2SIN(X)COS(X)
```

The Type of an Arithmetic Expression

The type of an arithmetic expression is determined by the types of its constituents. Suppose that $\mathcal{E}_i$ and $\mathcal{E}_j$ are integral expressions and that $\mathcal{E}_x$ and $\mathcal{E}_y$ are floating-point expressions. Further, take o to mean any of the arithmetic operators: + − · / or *. Then,

$\mathcal{E}_i$ o $\mathcal{E}_j$ is an integral expression, and

$\mathcal{E}_i$ o $\mathcal{E}_y$
$\mathcal{E}_x$ o $\mathcal{E}_j$ } are floating-point expressions.
$\mathcal{E}_x$ o $\mathcal{E}_y$

In general, if either expression is floating-point, then the result of combining them will be floating-point; if both expressions are integral, then the combination is also integral.

When mixed values are combined by the operators + − · and / , the compiler provides the program to convert the integral value to its corresponding floating-point form. The actual computation is done in floating-point. Exponentiation is usually performed by a routine taken from the library of the compiler. Separate entries to this routine are provided for each of the four combinations of integer and floating-point values.

If an integer constant appears in an expression of mixed types, the necessary conversion is performed at the time of compilation, resulting in no loss of efficiency in the object program. For example, if X is a floating-point variable, the expression X + 1 will be compiled as though the user had written X + 1.0.

Arithmetic Combinations of Integers

As mentioned in CHAPTER II, integers may be no more than ten digits in length. In all arithmetic operations, digits are dropped from the most significant end of the answer to produce a ten-digit result. Thus,

(734981)·(1000001) yields 4981734981;
5000000001 + 5000000001 yields 2.

Division of integers is unrounded. Thus,

3/2 yields 1; 7/11 yields 0; −41/3 yields −13.

Division by zero is undefined.

RELATIONS

Relations are provided to test the relative magnitudes of arithmetic quantities. These relations consist of two arithmetic expressions and a relational operator. A relation is one of the forms of *condition* used in conditional clauses and, as such, is employed to affect the sequence in which statements are executed.

GENERAL FORM:

$\mathcal{E}_1$ ρ $\mathcal{E}_2$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are arithmetic expressions, and ρ is a relational operator. This relation has the value ***true*** if $\mathcal{E}_1$ does indeed stand in the relation ρ to $\mathcal{E}_2$; it is otherwise ***false***. It is used only in control statements; (see IF, UNTIL, and EITHER IF). The relational operators employed in this compiler are GTR, GEQ, EQL, LEQ, LSS, and NEQ. Their significance is indicated in the following table.

RELATION†	CONVENTIONAL MATHEMATICAL NOTATION	MEANING
$\mathcal{E}_1$ GTR $\mathcal{E}_2$	$\mathcal{E}_1 > \mathcal{E}_2$	greater than
$\mathcal{E}_1$ GEQ $\mathcal{E}_2$	$\mathcal{E}_1 \geq \mathcal{E}_2$	greater than or equal to
$\mathcal{E}_1$ EQL $\mathcal{E}_2$	$\mathcal{E}_1 = \mathcal{E}_2$	equal to
$\mathcal{E}_1$ LEQ $\mathcal{E}_2$	$\mathcal{E}_1 \leq \mathcal{E}_2$	less than or equal to
$\mathcal{E}_1$ LSS $\mathcal{E}_2$	$\mathcal{E}_1 < \mathcal{E}_2$	less than
$\mathcal{E}_1$ NEQ $\mathcal{E}_2$	$\mathcal{E}_1 \neq \mathcal{E}_2$	not equal to

† Spaces are required to the left and right of the relational operator.

EXAMPLES:

```
X NEQ 0
ABS(L − LPRIME) LSS EPSILON
T GTR TMAX
```

Within the context of this compiler, two numbers are equal if the quantities which are their internal machine representations are identical. Ordinarily, it would suffice to say that two numbers are equal if their difference is zero. In the BURROUGHS 220 this may not be the case, since if A and B are floating-point numbers, and if $|A - B| < 0.1 \times 10^{-50}$, the result produced for their difference A − B will be zero.

The programmer must be responsible for determining whether he means that A and B are identical—in which case he uses A EQL B— or whether he means that their difference is zero, in which case he writes (A − B) EQL 0.

The compiler permits arithmetic operations to be performed on quantities which are not of the same type. In similar fashion, relational operators may be used to compare quantities which are not of the same type. If a floating-point quantity is to be compared to an integral quantity, the integer will be converted to its corresponding floating-point form prior to the comparison. If the integer is a constant, the conversion occurs during compilation.

If a relation is enclosed within parentheses, it becomes a Boolean expression which has a truth value of either one or zero, depending on whether the relation is ***true*** or ***false*** respectively.

BOOLEAN EXPRESSIONS

Boolean quantities may be combined by means of logical operators to form *Boolean expressions*, in a manner entirely analogous to the combining of arithmetic quantities by arithmetic operators. Boolean expressions are either ***true*** or ***false***, depending entirely on the truth values of the quantities entering into the expression and the definitions of the logical operators combining them.

Logical Operators

The logical operators which are accepted by the compiler are NOT, AND, OR, IMPL, and EQIV. These operators are called negation, conjunction, disjunction, implication, and equivalence, and are defined as follows (P and Q are Boolean quantities):

The expression NOT P is ***true*** whenever P itself is ***false***; it is ***false*** whenever P is ***true***.

The expression P AND Q is ***true*** if and only if both P and Q are ***true***. If either P or Q is ***false***, then P AND Q is also ***false***.

The expression P OR Q is ***true*** if either P or Q or both are ***true***. P OR Q is ***false*** only when both P and Q are ***false***.

The expression P IMPL Q is ***false*** only when P is ***true*** and Q is ***false***.

The expression P EQIV Q is ***true*** if P and Q are both ***true*** or both ***false***. If either P is ***true*** and Q is ***false*** or P is ***false*** and Q is ***true***, then P EQIV Q is ***false***.

These definitions are summarized in the following table.

P	Q	NOT P	P AND Q	P OR Q	P IMPL Q	P EQIV Q
false	false	true	false	false	true	true
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false

Construction of Boolean Expressions

Any variable, constant, or evaluated function will itself constitute a Boolean expression, if it is of Boolean type.

In addition, if $\mathcal{E}_1 \rho \mathcal{E}_2$ is an arithmetic relation, and $\mathcal{B}_1$ and $\mathcal{B}_2$ are any Boolean expressions, then each of the following is also a Boolean expression:

$(\mathcal{E}_1 \rho \mathcal{E}_2)$	$\mathcal{B}_1$ OR $\mathcal{B}_2$
$(\mathcal{B}_1)$	$\mathcal{B}_1$ EQIV $\mathcal{B}_2$
NOT $\mathcal{B}_1$	$\mathcal{B}_1$ IMPL $\mathcal{B}_2$
$\mathcal{B}_1$ AND $\mathcal{B}_2$	

The Boolean expression $(\mathcal{E}_1 \rho \mathcal{E}_2)$ may be combined with any of the logical operators to form more complex Boolean expressions.

Precedence of Logical Operators

Conventions for the order of precedence of logical operators are not so well established as are those for arithmetic operators. However, we shall assume the following order, which is apparently the most common:

Unless indicated otherwise by the use of parentheses, NOT *will be executed before* AND; AND *will be executed before* OR; OR *will be executed before* IMPL; *and* IMPL *will be executed before* EQIV.

The expression P IMPL Q IMPL R is ambiguous; parentheses should be used to express the exact meaning desired.

EXAMPLES:

```
NOT(P AND Q) OR R IMPL P OR NOT Q
NOT (NOT P) EQIV P
P IMPL P OR U AND V
(P OR Q) AND NOT (P AND Q)
(A LEQ X) AND (X LEQ B)
(ERROR LSS TOLERANCE) OR (N GTR 40)
R AND S OR (F(Z) EQL 4)
(M.N(R – 2) + 4 LSS TAN (BETA – ALPHA)) OR FLAG
(U.SINH(M) GTR M7) EQIV (V.COSH(M) GTR M12)
```

Any Boolean expression may appear in an arithmetic expression, where it will in all respects behave as if it were an integer taking on the values zero or one. In such a case, Boolean operations will be executed prior to arithmetic operations, unless parentheses have been used to specify otherwise.

EXAMPLES:

```
G – 0.3N.(D LSS 300)
A + V.NOT B1 OR B2
```

ASSIGNMENT STATEMENTS

GRAMMAR OF STATEMENTS

COMPOUND STATEMENTS

STATEMENT LABELS

IV. . .

statements

THE *statement*, S, is the fundamental unit of expression in the description of an algorithm. Most of what follows in this manual deals with the formation of statements and their interrelation to form larger constructs. *Statements specify something that the object program is to do. Declarations give information to the compiler about the program being compiled.* After this chapter, the word 'statement' will usually be employed to mean only a statement. However, for the present, 'statement' will stand for either a statement or declaration.

The first part of this chapter discusses one particular kind of statement—the assignment statement. The last part of the chapter deals with the grammar of statements in general, using assignment statements for examples.

THE ASSIGNMENT STATEMENT

The *assignment statement* specifies an expression which is to be evaluated and a variable which is to have the resulting value assigned to it.

GENERAL FORM:

$\mathcal{V} = \mathcal{E}$

where $\mathcal{V}$ is a variable and $\mathcal{E}$ is an expression. Note that the symbol = is used in a special sense in this compiler to signify the process of substitution. Thus X = X + 1 means 'using the current value of the variable X, evaluate the expression X + 1, and assign the result as the new value of X.' Although X = X + 1 is not a valid equation, it is a well-formed statement, and the compiler will carry out the indicated substitution. Thus the following valid algebraic expression

X*2 = Y + 2

has no meaning to the compiler, while

X = SQRT (K)

is a valid statement, and can be evaluated by a compiled program, which then assigns the value of $\sqrt{K}$ to the variable X.

Arithmetic Assignment Statements

If the variable $\mathcal{V}$ in $\mathcal{V} = \mathcal{E}$ is of integer or floating-point type, then we have an arithmetic assignment statement. If $\mathcal{V}$ is an integer and $\mathcal{E}$ is floating-point, then the value of $\mathcal{E}$ will be converted to integer form (truncating any fractional part) before the assignment is made. If $\mathcal{V}$ is floating-point and $\mathcal{E}$ is integral, then the value of $\mathcal{E}$ will be converted to the corresponding floating-point number. If $\mathcal{E}$ is Boolean, it is treated as if it were integral.

EXAMPLES:

```
R = (-B + SQRT(B*2 - 4A.C))/2A
FUNC = Y(I) + (Y(I + 1) - Y(I))(ARG - X(I))/(X(I + 1) - X(I))
U = X.COS(THETA) + Y.SIN(THETA)
OMEGA = 1/SQRT(L.C)
E = M.C*2
P(N) = ((2N - 1).P(N - 1) - (N - 1).P(N - 2))/N
C(I,J) = C(I,J) + A(I,K).B(K,J)
```

Boolean Assignment Statements

If the $\mathcal{V}$ in $\mathcal{V} = \mathcal{E}$ is a Boolean variable, we then have a Boolean assignment statement. In this case, *the expression $\mathcal{E}$ must be Boolean.*

EXAMPLES:

```
FLAG = (SWITCH4 OR SWITCH5) AND FLAGPRIME
TEST = (X NEQ 0) AND (Y NEQ 0)
M(I,J) = M(I,J) OR (K(I,K) AND K(J,K))
TOGGLE3 = TOGGLE4 AND TAG OR (U LSS V)
```

Generalized Assignment Statement

GENERAL FORM:

$\mathcal{V}_1 = \mathcal{V}_2 = \cdots = \mathcal{V}_n = \mathcal{E}$

If it is desired to assign the same value to a number of variables, it can be accomplished in a single statement by employing this generalized form.

Note that if the list of variables to which a value is being assigned is of mixed type, then conversion of type will be performed; e.g., assume X, Y, and $\mathcal{E}$ are floating, and I is integer. Then the statement

$X = I = Y = \mathcal{E}$

will cause $\mathcal{E}$ to be truncated to an integer before storing into I, and this truncated result floated before storing into X. Thus, in this example, $X = I = Y = \mathcal{E}$, $X = Y = I = \mathcal{E}$, and $I = X = Y = \mathcal{E}$ may all give different results when $\mathcal{E}$ is floating.

EXAMPLES:

```
V = X = Y = 15.302
A(I) = B(I) = Z = 0
```

THE GRAMMAR OF STATEMENTS

This section discusses certain definitions and rules of the compiler language which have to do with the writing of statements. The basic rule of the grammar of statements is that *statements must be separated by semicolons.*

Even though a statement ends on a given line and the next statement begins on the next line, the separating semicolon must be indicated. *The end of a line has no meaning as punctuation.*

GENERAL FORM:

$\cdots\ \mathcal{S};\ \mathcal{S}\ \cdots$

where the symbol $\mathcal{S}$ represents any statement. Unless otherwise indicated, statements are performed one after the other in the sequence in which they are written. As many statements as desired may be written on a line (subject of course to the physical limitations of the input medium), or a statement may use as many lines as are required for its expression.

EXAMPLE:

```
W = A + B; X = A - B; Y = A.B; Z = A/B
```

Compound Statements

It is frequently desirable to group several statements together to form a larger construct which is to be considered as a single statement. Such a construct is called a *compound statement.*

GENERAL FORM:

BEGIN $\mathcal{S}_1; \mathcal{S}_2; \cdots; \mathcal{S}_n$ END

where $\mathcal{S}_1$ through $\mathcal{S}_n$ are statements. The words BEGIN and END serve as opening and closing 'statement parentheses.' Indeed, the symbols '(' and ')' may be substituted for the words BEGIN and END with no change in meaning.

Throughout this description of the compiler, unless the contrary is specifically stated, the word 'statement' and the symbol $\mathcal{S}$ should be construed to mean either a simple or a compound statement.

Certain other constructs involving the grouping of several statements automatically constitute compound statements. These will be discussed further in their proper context in CHAPTER V.

EXAMPLES:

```
BEGIN U = -B/2A; V = SQRT(U*2 - C/A);
R1 = U + V; R2 = U - V END

BEGIN S = SIN(THETA); C = COS(THETA);
XI = C.X + S.Y; ETA = -S.X + C.Y END

(S = A(I,J); A(I,J) = A(J,I); A(J,I) = S)
```

Labeling of Statements

It is often necessary to attach a name to a statement. This name is called a *statement label.* A statement label may be an identifier or an integer. (Leading zeros of an integer used as a statement label are without meaning to the compiler—the labels 13 and 013 are in all ways equivalent.) When a statement is to be identified, it is preceded by a label and the '..' separator.

GENERAL FORMS:

First form:

$\mathcal{I}$.. $\mathcal{S}$

Second form:

$\mathcal{N}_1$.. $\mathcal{S}$

where $\mathcal{I}$ is an identifier, $\mathcal{N}_1$ is an integer, and $\mathcal{S}$ is any statement. These forms declare the identifier or integer to be a label.

EXAMPLES:

```
START..SUM = 0
LEGENDRE..P(N) = ((2N - 1) P(N - 1) - (N - 1)P(N - 2))/N
ROTATE..BEGIN S = SIN(THETA); C = COS(THETA);
XI = C.X + S.Y; ETA = -S.X + C.Y END
27..BETA = ARCTAN(HORIZ/VERTL) - ALPHA
```

When labeling a compound statement, the programmer may repeat the statement label after the word END.

This may be done for readability of the print-out produced during compilation; the compiler itself makes no use of the information.

GENERAL FORM:

£..BEGIN S_1; S_2; ⋯; S_n END £

This technique of demarcation of compound statements is useful when such statements are nested.

EXAMPLES:

```
ROOTS..BEGIN U = -B/2A; V = SQRT(B*2 - 4.A.C)/2A;
R1 = U + V; R2 = U - V END ROOTS
```

In those cases where it is necessary during the running of a program to transfer from a point in a BEGIN ⋯ END clause to a point just before the word END, a labeled dummy statement is employed.

GENERAL FORM:

BEGIN S_1; S_2; ⋯; S_n; £..END

This statement, which does not in itself produce any action, takes the form of a label followed by two periods, directly preceding the word END which terminates the group of statements. An example of this is shown in CHAPTER VI, SEARCH OF A RECTANGULAR GAME FOR A SADDLE POINT.

It is sometimes necessary to introduce a section of machine-language coding into the compiled program, which will then act in all respects like a statement. To do this, one employs the declarator EXTERNAL STATEMENT in the symbolic program.

GENERAL FORM:

EXTERNAL STATEMENT £

The identifier £ serves as the label of the EXTERNAL STATEMENT.

The definition of the statement, i.e., the machine-language program itself, appears after the FINISH declarator of the symbolic program. (See APPENDIX F, *Construction of Machine-Language Programs.*)

TYPE
ARRAY
COMMENT
FINISH

V...

basic declarations

THE DECLARATIONS OF TYPE—FLOATING, REAL, INTEGER, and BOOLEAN are defined in this chapter, together with the ARRAY, COMMENT, and FINISH declarations. These do not exhaust the entire set of declarations available to the programmer; however, the others constitute separate subjects in themselves and are therefore reserved for later chapters.

Declarations determine how the compiled program will treat certain of its elements. It is thus necessary to precede the use of an element with such a declaration.

DECLARATIONS OF TYPE

Declarations of type are used to indicate that a specified set of identifiers represent quantities of a given type (floating-point, integer, or Boolean). By the use of prefixes, entire classes of identifiers are declared to be of a given type. In addition, it is possible to declare that a variable not appearing in any declaration of type is of a given type.

Constructions of Declarations of Type

GENERAL FORM:

FLOATING 𝔗
REAL 𝔗
INTEGER 𝔗
BOOLEAN 𝔗

where 𝔗 is a *type list* to be defined below. These statements declare the identifiers given in 𝔗 to be of floating-point, integer, and Boolean types. (FLOATING and REAL produce equivalent results in the compiler.) A *type list* consists of a sequence of entries separated by commas. Possible entries include identifiers, identifiers followed by blank subscripts, and prefixes.

EXAMPLE:

```
INTEGER I, J, K, L, Z, GCD(,), TABLE( )
```

(Note that the use of a parenthesis pair following an identifier in a declaration of type has no effect on the compiled program; the parentheses are there only for the convenience of the programmer. Also, the entire type list may be enclosed in parentheses without affecting the object program.)

EXAMPLE:

```
BOOLEAN (SW1, SW2, FLAG, TOGGLE(,,))
```

The Use of Prefixes

If desired, one may put a prefix into a type list rather than use an identifier. A *prefix* consists of an identifier of no more than five characters, followed by three periods.

GENERAL FORM:

𝔤...

EXAMPLE:

```
MQR4...
```

The appearance of this prefix in a declaration of type means that any variable, function, or array, the identifier of which has MQR4 as its first four characters, and which is not explicitly declared, is of the specified type.

It is possible for prefixes to introduce apparent ambiguities. Consider, for example,

```
FLOATING ABCD, AB4; INTEGER AB...;
BOOLEAN ABC...
```

What are the types of AB13, ABCD, ABCDEF, AB4, and AB5? The rule governing this situation is: *Unless specifically indicated in a type list, an identifier is matched against the longest applicable prefix.* Thus, the above identifiers are of integer, floating, Boolean, floating, and integer types, respectively.

Declaration by Default

The word OTHERWISE may be written in lieu of a type list. This form indicates that any name of a variable, array, or function not specifically declared and not matching any of the prefixes is to be of the type denoted by this declaration. If no such declaration is given, any undeclared variable, array, or function will be assumed to be floating-point. This construction may be called *declaration by default.*

EXAMPLES:

```
BOOLEAN SW, P,Q,R; FLOATING X,Y,Z,F( );
  INTEGER OTHERWISE

INTEGER I, J, K, N, M..., G; BOOLEAN OTHERWISE
```

To repeat the remark made in the introduction of this chapter: *The type of an identifier must be declared before that identifier is used in any other statement. If an identifier is used prior to a declaration of type it is declared, by default, as* FLOATING.

THE ARRAY DECLARATION

The ARRAY declaration provides a means of referring to a collection of values by the use of a single identifier, and at the same time specifies to the compiler the structure which is to be imposed on this collection.

Arrays in this compiler are restricted to those of rectangular construction in *n*-dimensional space.

If the identifier of an array is declared in a declaration of type, then that declaration of type must precede the ARRAY declaration.

Construction of ARRAY Declarations

An array must have been described by an ARRAY declaration prior to the use of any subscripted variable which represents an element of that array.

GENERAL FORM:

ARRAY $\mathfrak{L}\mathfrak{g}$, $\mathfrak{L}\mathfrak{g}$, ..., $\mathfrak{L}\mathfrak{g}$

where $\mathfrak{L}\mathfrak{g}$'s are list items of the array declarator list. These list items take on two general forms:

First form:

$\mathfrak{g}\ (\mathfrak{N}_1, \ldots, \mathfrak{N}_q)$

Second form:

$\mathfrak{g}\ (\mathfrak{N}_1, \ldots, \mathfrak{N}_q) = (\mathfrak{M}, \mathfrak{M}, \mathfrak{M}, \ldots, \mathfrak{M})$

Both of these forms declare $\mathfrak{g}$ to be an array of q dimensions. Each dimension contains the number of elements given by the corresponding value of $\mathfrak{N}$; hence the value of $\mathfrak{N}$ is the maximum which a corresponding subscript expression may assume.

The second form is used to set initial values for the elements of the array at load time; see *Filling an Array*, below.

EXAMPLE:

```
ARRAY (M(3,4), CHAR (6,6,6), VECTOR (100))
```

This declaration reserves twelve cells in storage for the two-dimensional array M, 216 cells for the three-dimensional array CHAR, and 100 cells for the one-dimensional array VECTOR.

At the programmer's option, the list of arrays being declared may be enclosed in parentheses to improve the readability of the symbolic program. Such use of these parentheses will have no effect on the compilation.

Filling an Array

An item in an ARRAY declaration may have appended to it a list of values to be assigned at the beginning of computation to the elements of the array. Referring to the second form, above:

Second form:

$\mathfrak{g}\ (\mathfrak{N}_1, \ldots, \mathfrak{N}_q) = (\mathfrak{M}, \mathfrak{M}, \mathfrak{M}, \ldots, \mathfrak{M})$

The quantities $\mathfrak{M}$ are constants (with their respective signs) which are placed in the positions of the array $\mathfrak{g}$. (The compiler will, if necessary, convert these constants at the time the program is compiled to agree in type with that of the identifier $\mathfrak{g}$.)

EXAMPLE:

```
ARRAY Q (3,2) = (7.3, 9.1, 4, 127.3, +4.19, –2.2)
```

Assuming Q has been previously declared to be floating, this declaration will cause the matrix Q

$$\begin{bmatrix} 7.3 & 9.1 \\ 4.0 & 127.3 \\ 4.19 & -2.2 \end{bmatrix}$$

to be available in memory when the compiled program is loaded.

It is not necessary to fill up the entire array in this manner. Cells to which no constant is assigned are cleared to zero at the time the program is loaded from tape.

Referring again to the second form above, the constants $\mathfrak{M}$ are placed in the array $\mathfrak{g}$ in the following order:

The initial subscript is (1, 1, ..., 1). For each succeeding value of $\mathfrak{M}$ the rightmost subscript is advanced by one.

After the rightmost subscript reaches its maximum value, $\mathfrak{N}_q$, it is reset to one, and at the same time the subscript to its left is advanced by one.

In similar fashion, the other subscripts are advanced, the subscript in the $(i-1)$th position being increased by one at the same time that the ith subscript is reset to one.

These cycles continue until all data have been stored, or until all subscripts have reached their respective maximum values.

Assuming an array $A(n_1, n_2, ..., n_{q-1}, n_q)$, this results in the following sequence of subscripts:

CYCLE	SUBSCRIPT
First	1, 1, ..., 1, 1 1, 1, ..., 1, 2 . . . . . . 1, 1, ..., 1, n_q
Second	1, 1, ..., 2, 1 1, 1, ..., 2, 2 . . . . . . 1, 1, ..., 2, n_q
Third	1, 1, ..., 3, 1
.	.
.	.
.	.
$(n_1 \cdot n_2 \cdot \ ... \ \cdot n_{q-1} - 1)$th	$n_1, n_2, ..., n_{q-1} - 1, 1$ $n_1, n_2, ..., n_{q-1} - 1, 2$ . . . . . . $n_1, n_2, ..., n_{q-1} - 1, n_q$
$(n_1 \cdot n_2 \cdot \ ... \ \cdot n_{q-1})$th	$n_1, n_2, ..., n_{q-1}, 1$ $n_1, n_2, ..., n_{q-1}, 2$ . . . . . . $n_1, n_2, ..., n_{q-1}, n_q$

THE COMMENT DECLARATION

The COMMENT declaration allows the programmer to include any clarifying remarks, identifying symbols, etc., in the printed compilation. The COMMENT declaration does not appear as part of the compiled program, and has no effect on the program; it merely sets apart any string of characters for printing as part of the compilation. Since the comment extends to the next semicolon, a semicolon obviously cannot be used within the string of characters.

GENERAL FORM:

COMMENT s

where s is any string of characters *not containing a semicolon.*

EXAMPLE:

```
COMMENT SMOOTH FIELD DATA AND REDUCE TO
  STANDARD FORM
```

There is one restriction on the use of the COMMENT declaration. *It must not be the last statement of a compound statement*; that is, a COMMENT statement must not be terminated by an 'END' or a ')'. Only a ';' may follow the comment.

THE FINISH DECLARATION

The FINISH declaration defines the end of the program being compiled, and terminates the compilation. A FINISH declaration must appear as the last statement in any program and may appear nowhere else in the symbolic program. The semicolon following a FINISH declaration is essential; it may not be omitted.

GENERAL FORM:

FINISH;

EXAMPLE:

```
FINISH;
```

(See APPENDIX B for the manner in which the compiler treats the FINISH declaration.)

TRANSFER OF CONTROL
SUSPENSION OF COMPUTATION
CONDITIONAL EXECUTION
CONTROL OF ITERATIONS

VI . . .

basic control statements

THIS CHAPTER deals with the means of expressing the 'flow of control' of an algorithm which has been described in compiler language. The order of evaluation of equations is as important to the description of an algorithm as are the equations themselves. Experience has shown that there is a relatively small number of constructions which commonly appear in the description of algorithms. This group of constructions has been included in the compiler language.

The basic control statements provide the abilities:

First, to transfer control to another part of the problem (the GO TO and SWITCH statements);

second, to suspend computation (the STOP statement);

third, to execute statements contingent on given criteria (the IF and alternative statements); and

fourth, to control iterative processes (the FOR and UNTIL statements).

TRANSFER OF CONTROL

The GO TO Statement

The GO TO statement provides the ability to transfer control from one part of the compiled program to another.

GENERAL FORM:

GO TO £

where £ is a statement label.

The statement with the label £ will be executed immediately after the GO TO statement. The word TO is redundant and may be omitted.

EXAMPLES:

```
GO TO START
GO A16
GO TO 14
GO LOOP
```

The SWITCH Statement

A second method of transferring control is provided by the SWITCH statement. The SWITCH statement uses the value of an expression to select one entry from a list of statement labels, and then transfers control to the statement bearing that label.

GENERAL FORM:

SWITCH $\mathcal{E}$, ($£_1$, $£_2$, $£_3$, ⋯, $£_n$)

where $\mathcal{E}$ is an arithmetic expression and $£_1$ through $£_n$ are statement labels.

The action of a SWITCH statement is as follows:

Let i equal the integral part of the expression $\mathcal{E}$; then,

If $i = 0$, the SWITCH statement has no effect, and control continues in sequence.

If $| i | \leq n$, then a transfer of control is made to the statement the label of which is in the ith position in the list.

If $| i | \geq n + 1$, then the action of the SWITCH statement is undefined.

EXAMPLES:

```
SWITCH Y + 2, (A1, A2, A3)
```

If Y + 2 = 0, no transfer occurs.
If Y + 2 = 1, transfer to the statement labeled A1.
If Y + 2 = 2, transfer to the statement labeled A2.
If Y + 2 = 3, transfer to the statement labeled A3.

```
SWITCH 31 + J, (XA, XB, XC, YA, YB, YC)
SWITCH MOD(K,4) + 1, (ALPHA, BETA, GAMMA, DELTA)
```

SUSPENSION OF COMPUTATION

The STOP Statement

The STOP statement serves to indicate the end of operation or a temporary halt in a compiled program. (Computation is resumed with the next statement in sequence when the START switch is depressed.) If desired, a STOP statement may be accompanied by an expression the value of which is displayed in the A register when the computer stops.

GENERAL FORMS:

First form:

STOP

Second form:

STOP $\mathcal{E}$

where $\mathcal{E}$ is any expression.

In the case of the first form, the computer stops with the contents of the A register undefined; in the case of the second form, the value of the expression $\mathcal{E}$ is found in the A register.

EXAMPLES:

```
STOP
STOP 4241535362
STOP ANSWER(J)
```

CLAUSES

The GO TO, SWITCH, and STOP statements discussed thus far are by themselves complete statements and depend in no way on other statements to complete their meaning. The remainder of this chapter will discuss statements which bear an analogy to the dependent clauses of a natural language. In all cases, these clauses affect the behavior of the statement (or compound statement) which follows them.

A construction consisting of one or more clauses $\mathcal{C}$ followed by a statement $\mathcal{S}$,

$\mathcal{C}_1; \mathcal{C}_2; \mathcal{C}_3; \sim; \mathcal{S}$

is in itself a compound statement, requiring no additional punctuation.

Of course, if a clause is to affect the behavior of several statements, those statements must be grouped together as a compound statement:

$\mathcal{C}$; BEGIN $\mathcal{S}_1; \mathcal{S}_2; \mathcal{S}_3; \sim \mathcal{S}_n$ END

Examples of these constructions will be given in context below.

CONDITIONAL EXECUTION

The IF Statement

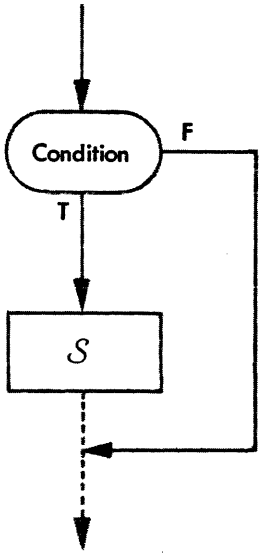
The IF statement consists of an IF clause followed by a semicolon and then by a statement. An IF clause provides the means of indicating that the next statement in sequence is to be conditionally executed. The clause consists of the reserved word IF followed by a condition which may be either a Boolean expression or a relation.

GENERAL FORM:

IF $\mathcal{B}$; $\mathcal{S}$

where $\mathcal{B}$ is a condition and $\mathcal{S}$ is any statement. Note that the semicolon is used to separate a clause from its associated statement.

The action of the IF statement is described graphically by means of the following flow chart:



If the condition of an IF clause is a Boolean expression with truth value one, or if it is a relation made up of expressions which do stand in the given relation to one another, then statement $\mathcal{S}$ will be executed; otherwise, statement $\mathcal{S}$ will be skipped over and control continued in sequence following statement $\mathcal{S}$.

EXAMPLES:

```
IF (X*2 GTR 7); STOP
IF (I EQL J); A(I,J) = 1
IF (M NEQ 0) OR (N NEQ 0); GO TO LAST
IF P EQIV R OR P EQIV S; K = B(J)
IF (X LEQ 0) AND FLAG; X = ABS(X)
IF U OR V AND (X LSS 2.4); BEGIN U = 0;
  V = 0; GO TO REPEAT END
```

The most common form of condition which appears in an algorithm is a simple comparison between the magnitudes of two arithmetic quantities. While this situation is certainly provided for under the form of Boolean expressions (see the first two examples above), the slightly more concise relations are also allowed. In those cases where a relation is applicable and is used, the result will be the compilation of a significantly more efficient object program. The use of a relation in an IF clause is recommended wherever possible.

EXAMPLES:

```
IF X*2 GTR 7; STOP
IF I EQL J; A(I,J) = 1
IF I EQL IX; SWITCH IX, (A,B,C)
IF ABS(TERM) LSS EPSILON; GO OUT
IF TGL4; IF Z GTR X + Y*2 - 4; BEGIN Z = Y - 1;
  X = Y/(U + Y); GO LOOP4 END
```

The Alternative Statement

An extended form of the IF statement is provided by the *alternative statement.* A sequence of conditions is examined—in order—until one is found which is satisfied. A statement associated with that particular condition is then executed; the remainder of the alternatives are ignored. A second form indicates a statement to be executed in the event that none of the sequence of conditions is satisfied.

GENERAL FORMS:

First form:

EITHER IF $\mathcal{B}_1$; $\mathcal{S}_1$; OR IF $\mathcal{B}_2$; $\mathcal{S}_2$; ⋯;
OR IF $\mathcal{B}_n$; $\mathcal{S}_n$ END

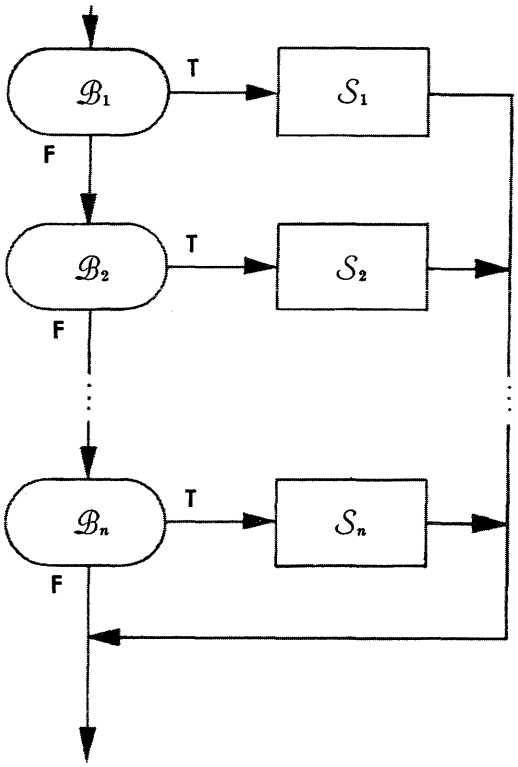
Second form:

EITHER IF $\mathcal{B}_1$; $\mathcal{S}_1$; OR IF $\mathcal{B}_2$; $\mathcal{S}_2$; ⋯;
OR IF $\mathcal{B}_n$; $\mathcal{S}_n$; OTHERWISE; $\mathcal{S}_{n+1}$

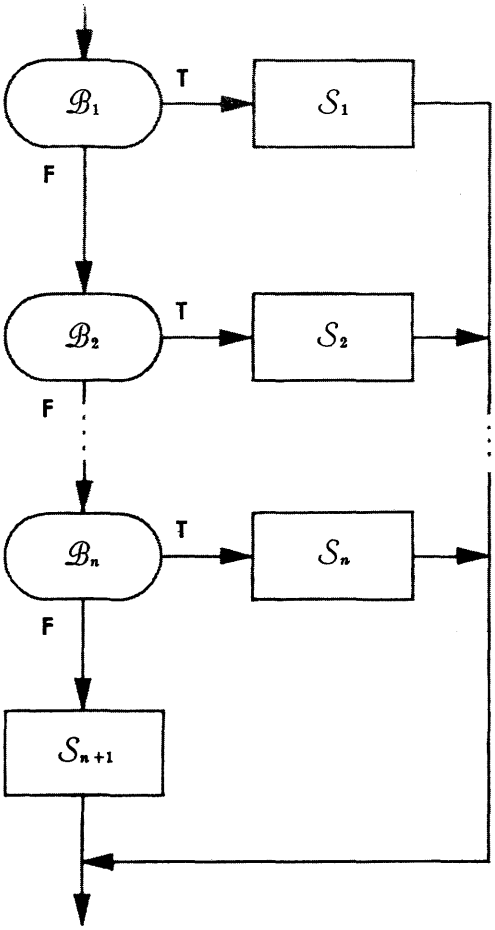
Whenever one of the alternative statements is found to be ***true***, the statement associated with OTHERWISE will not be executed.

Any of the conditions $\mathcal{B}$ may be replaced by the simpler form, $\mathcal{E}_1 \rho \mathcal{E}_2$, whenever desired.

The following flow charts will serve to explain these two statements more precisely. For the first form we have:



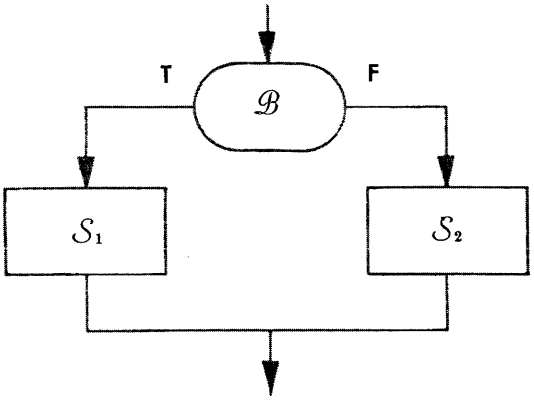
For the second form we have:



In the second form, no OR IF clauses need be used. If no OR IF clauses are used, the alternative statement becomes

EITHER IF $\mathcal{B}$; $\mathcal{S}_1$; OTHERWISE; $\mathcal{S}_2$

which expresses the very common construction:



EXAMPLE:

```
COMMENT EVALUATE POLYNOMIAL
USING RECURSION RELATION EMPLOYING PREVIOUS
VALUES WHEN POSSIBLE. N IS ORDER OF POLYNOMIAL
AND X IS ARGUMENT;
EITHER IF N EQL 0; BEGIN M = 1.0**40;
PN1 = 1.0 END;
OR IF N EQL 1; BEGIN M = 3; Z = X + X;
R = X + Z; PN2 = 1; PN1 = X END;
OR IF (X + X EQL Z) AND (N GEQ M - 1);
GO TO RECURSE;
OTHERWISE; BEGIN PN2 = 1; PN1 = X;
Z = X + X; R = X + Z; M = 3;
RECURSE.. BEGIN N = N + 1; FOR M = (M, 1, N);
BEGIN R = R + Z; PN = (R.PN1 - (M - 1) PN2)/M;
PN2 = PN1; PN1 = PN END END RECURSE END;
POLYNOMIAL = PN1
```

Nested IF Statements

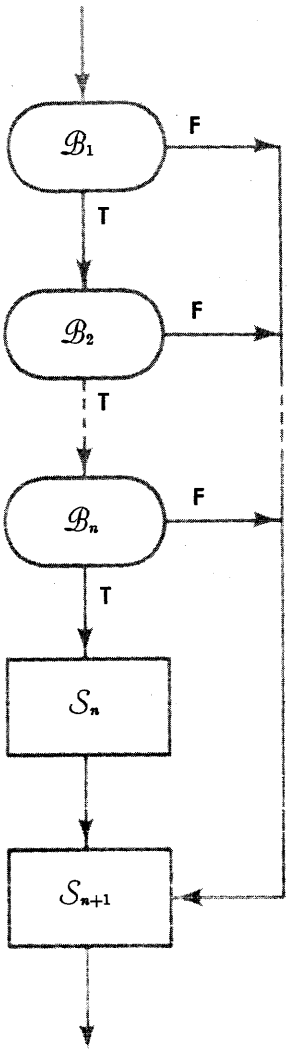
The nesting of IF statements results in a construct somewhat similar to that of the alternative statement.

GENERAL FORM:

IF $\mathcal{B}_1$; IF $\mathcal{B}_2$; $\cdots$; IF $\mathcal{B}_n$; $\mathcal{S}_n$; $\mathcal{S}_{n+1}$

where $\mathcal{B}_i$ is a Boolean or relational expression.

The statement $\mathcal{S}_n$ associated with the last condition is executed if and only if all the conditions are found sequentially to be ***true***. The first unsatisfied condition will cause the remainder of these conditions to be ignored and the statement $\mathcal{S}_{n+1}$ to be executed. The following flow chart illustrates the logic of the construct:



EXAMPLE:

```
IF MOD (YR,4) EQL 0; IF MO EQL 2;
IF DAY EQL 29; GO TO LEAPYEAR;
WRITE (;; VALIDATE)
```

CONTROL OF ITERATIONS

The UNTIL Statement

The UNTIL statement is used primarily to provide control of iterative processes where escape from the loop depends upon a result calculated within the loop. An UNTIL statement consists of an UNTIL clause followed by a semicolon, followed by a statement.

GENERAL FORMS:

First form:

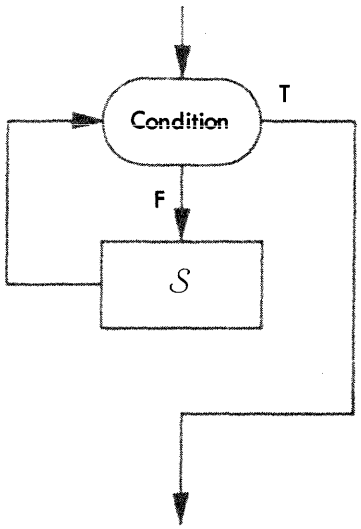
UNTIL $\mathcal{B}$; $\mathcal{S}$

Second form:

UNTIL $\mathcal{S}_1$ ρ $\mathcal{S}_2$; $\mathcal{S}$

where $\mathfrak{B}$ is any Boolean expression, $\mathcal{E}_1 \rho \mathcal{E}_2$ is a relation, and $\mathcal{S}$ is any statement.

The action of the UNTIL statement is described graphically by means of the following flow chart:



First form:

The statement $\mathcal{S}$ is executed repetitively until the Boolean condition is satisfied; control then continues in sequence. If $\mathfrak{B}$ is satisfied initially, $\mathcal{S}$ will not be executed at all.

EXAMPLES:

```
COMMENT TABLE LOOK-UP;
I = 1; UNTIL (T(I) GEQ ARGMT); I = I + 1
```

```
UNTIL (L – LPRIME LSS TOLERANCE) AND
  (V LSS 0.01); BEGIN V = V*2; LPRIME = L;
  L = ITER (L,V) END
```

```
COMMENT SUM SERIES FOR E**–X;
N = 1; E = 1; T = 1; UNTIL (ABS(T) LSS 1**–6) OR
  (N GTR 30); BEGIN T = –T.X/N; E = E + T;
  N = N + 1 END
```

Second form:

As in the case of the IF statement, the UNTIL statement is provided with an alternate form to produce a more efficient object program in those cases where the condition to be tested consists of a relation between two arithmetic quantities.

EXAMPLES:

```
COMMENT SEARCH FOR ROOT OF F( );
FA = F(A); UNTIL ABS (B – A) LSS EPS;
BEGIN U = (A + B)/2; EITHER IF FA.F(U) GTR 0; A = U;
OTHERWISE; B = U END
```

```
COMMENT ITERATED TRAPEZOIDAL INTEGRATION;
H = (B – A); I = (F(A) + F(B))/2; J = 0;
UNTIL I – 2 J LSS 0.003H;
  BEGIN Q = H; J = I; H = H/2; X = A + H;
  UNTIL X GTR B;
  BEGIN I = I + F(X); X = X + Q END END
```

The FOR Statement

The FOR statement finds its principal use in the control of an iteration where the statement or statement group to be iterated involves a variable (the *induction variable*) which must take on a succession of values. It is also used to cause a statement to be executed a predetermined number of times.

A FOR statement is comprised of a FOR clause and its associated statement, which must be separated from the clause by a semicolon.

GENERAL FORM:

FOR $\mathcal{V} = \mathcal{IL}$; $\mathcal{S}$

where $\mathcal{V}$ is a variable, $\mathcal{IL}$ is an iteration list, and the $\mathcal{S}$ is any statement.

The iteration list describes the sequence of values that the variable $\mathcal{V}$ is to assume. The statement $\mathcal{S}$ will be executed for each of these values. After the iteration list has been exhausted, the statement following $\mathcal{S}$ will be executed. Note that a GO TO statement might transfer control out of the FOR loop before the iteration loop is exhausted.

The most common form that an iteration assumes is a triplet of expressions separated by commas and enclosed in parentheses.

First form:

($\mathcal{E}_I$, $\mathcal{E}_S$, $\mathcal{E}_T$)

where $\mathcal{E}_I$, $\mathcal{E}_S$, and $\mathcal{E}_T$ are arithmetic expressions.

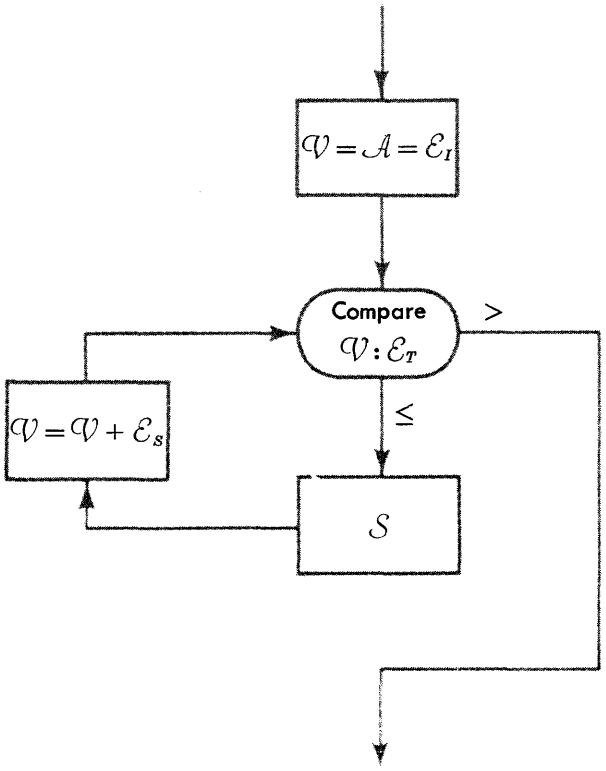
In this case, the FOR statement takes on the form:

FOR $\mathcal{V} = (\mathcal{E}_I, \mathcal{E}_S, \mathcal{E}_T)$; $\mathcal{S}$

If the first character of $\mathcal{E}_S$ is not a minus sign, then the form is equivalent to the simpler statements:

$\mathcal{V} = \mathcal{E}_I$; $\mathcal{L}$.. IF $\mathcal{V}$ LEQ $\mathcal{E}_T$;
BEGIN $\mathcal{S}$; $\mathcal{V} = \mathcal{V} + \mathcal{E}_S$; GO TO $\mathcal{L}$ END

The logical flow of this construct may be described diagrammatically as follows:



In the case that the first character of $\mathcal{E}_S$ is a minus sign, the FOR statement is equivalent to:

$\mathcal{V} = \mathcal{A} = \mathcal{E}_I$; £.. IF $\mathcal{V}$ GEQ $\mathcal{E}_T$;
BEGIN $\mathcal{S}$; $\mathcal{V} = \mathcal{V} + \mathcal{E}_S$; GO TO £ END

and the preceding flow chart holds if we replace > by <, and ≤ by ≥.

Note that if the test fails initially (i.e., $\mathcal{E}_I > \mathcal{E}_T$ in the first case, or $\mathcal{E}_I < \mathcal{E}_T$ in the second), the triplet is considered vacuous, and the statement $\mathcal{S}$ will not be executed at all. On exit from the FOR statement, the value of the induction variable $\mathcal{V}$ is that which it has when the test first failed.

The use of a GO TO or SWITCH statement to transfer control from outside the scope of the FOR loop to any labeled statement included inside the compound statement $\mathcal{S}$ may produce anomalous results.

EXAMPLES:

```
COMMENT EVALUATE INNER PRODUCT OF U( ) AND V( );
  DOT = 0; FOR I = (1,1,N);
  DOT = DOT + U(I).V(I)
```

```
COMMENT SEARCH RECTANGULAR GAME FOR
  SADDLE POINT;
FOR I = (1,1,M); BEGIN L = 0;
  FOR J = (1,1,N); IF A(I,J) GTR L;
  BEGIN L = A(I,J); T = J END;
  FOR K = (1,1,M); IF A(K,T) LSS L; GO AGAIN;
  GO FOUND; AGAIN.. END; GO NONE
```

```
COMMENT SOLVE EQUATIONS A(N X (N + 1)) FOR X( );
FOR K = (N + 1, -1, 1); BEGIN
  FOR I = (1,1,N); X(I) = A(I,1);
  FOR J = (2,1,K); BEGIN
    D = A(1,J)/X(1);
    FOR I = (2,1,N); A(I - 1, J - 1) = A(I,J) - X(I).D;
    A(N,J) = D END END
```

The second form that an iteration list may assume is a list of expressions separated by commas.

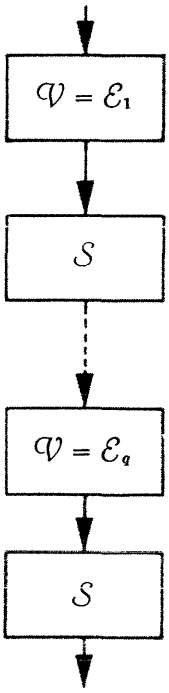
Second form:

$\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots, \mathcal{E}_q$

In this case, the FOR statement appears as

FOR $\mathcal{V} = \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots, \mathcal{E}_q$; $\mathcal{S}$

The behavior of this statement may be clarified by the following flow chart:



That is, the variable $\mathcal{V}$ is successively given the values of $\mathcal{E}_1$, $\mathcal{E}_2$, and so on through $\mathcal{E}_q$. The statement $\mathcal{S}$ is executed once for each value which $\mathcal{V}$ assumes.

EXAMPLES:

```
FOR PRIME = 2,3,5,7,11,13,17; S
FOR X = 0, 0.1, 0.5, 1.0, 5.0, 10.0; S
```

Third form:

The *third form* of an iteration list is actually a combination of the first two; triplets of expressions which appear in the first form may be used as members of the list of the second form. The sequence of values which results is the expected one.

EXAMPLES:

```
FOR Z = (0,1,10), (15,5,50), (100,50,1000), 5000, 10000; S
```

This statement would cause S to be executed for Z = 0, 1, 2, ..., 9, 10, 15, 20, 25, ..., 50, 100, 150, 200, 250, ..., 900, 950, 1000, 5000, and 10000.

```
FOR I = (1,1,N); FOR J = (1,1,I - 1), (I + 1,1,N);
  A(I,J) = A(I,J)/A(I,I)
```

This statement will divide off-diagonal elements of each row of matrix A(,) by the diagonal element of that row. Note that the first triplet of the second FOR clause is vacuous when I = 1; the second is vacuous when I = N.

SUBROUTINES
FUNCTIONS
INTRINSIC FUNCTIONS
PROCEDURES
EXTERNAL PROCEDURES

VII . . .

subprograms

ONE OF THE MOST IMPORTANT aspects of the stored-program computing device is its ability to treat subprograms which may be executed from any point in the main program. The compiler language includes several methods of defining subprograms, each of which has its particular field of application.

The declarations SUBROUTINE, FUNCTION, and PROCEDURE will be discussed, as well as the ENTER statement, the RETURN statement, a variation of the assignment statement, and the procedure-call statement. These declarations and statements are peculiar to the definition and use of subprograms.

SUBROUTINES

The form of the subprogram which is conceptually the simplest consists merely of a compound statement, which may be executed on demand from any part of the remainder of the program without the necessity of rewriting the actual compound statement each time its particular effect is desired. For our purposes here, such a compound statement will be called a *subroutine.*

The SUBROUTINE Declaration

The SUBROUTINE declaration states that the following compound statement represents a subroutine.

GENERAL FORM:

SUBROUTINE $\mathcal{I}$; BEGIN S_1; S_2; ⋯; S_n END

where $\mathcal{I}$ is an identifier and S_1 through S_n are the statements which define the effect of the subroutine. The identifier becomes the subroutine label. In such a case, $\mathcal{I}$ is not a statement label in the usual sense although it may—at the programmer's option—follow the word END, as may a label of any compound statement.

All the identifiers which appear in a subroutine have precisely the same meanings as those assigned to them outside the subroutine. This is what is meant when a subroutine is said to be *dependent* on the program in which it is defined.

The RETURN Statement

The compound statement which defines the subroutine is executed starting with its first component statement. One (or more) of the statements S_1 through S_n which compose the subroutine and which follow the SUBROUTINE declaration must be a RETURN statement. Computation within the subroutine proceeds until a RETURN statement is encountered.

GENERAL FORM:

RETURN

The RETURN statement causes control again to be resumed in sequence at that point at which the subroutine was called. If control is to be returned to the point following the entry to the subroutine, the RETURN statement must be used. 'Running off the end' of a subroutine will produce anomalous results. Exit may of course be made from a subroutine through the use of a GO TO or SWITCH statement.

A subroutine need not be defined prior to its use. Subroutines may be defined within other subroutines.

EXAMPLE:

```
SUBROUTINE EVALUATE; BEGIN U = 0; V = 0;
  FOR I = (1,1,N); FOR J = (1,1,N); BEGIN W = 0;
  FOR K = (1,1,N); W = W + X(I,K).Y(K,J); IF I EQL J;
  W = W - 1; U = U + ABS(W); V = MAX(V,ABS(W)) END;
  U = U/N*2; RETURN END EVALUATE
```

The ENTER Statement

The ENTER statement is used to initiate the execution of a subroutine (*to call* a subroutine).

GENERAL FORM:

ENTER $\mathcal{G}$

where $\mathcal{G}$ is the subroutine label.

EXAMPLE:

```
SUBROUTINE CHEBYSHEV;
BEGIN EITHER IF N EQL 0; (M = 1.0**40; PN1 = 1.0);
OR IF N EQL 1; (M = 2; Z = X + X; PN2 = 1; PN1 = X);
OR IF (X + X EQL Z) AND (N GEQ M); ENTER RECURSE;
OTHERWISE; BEGIN PN2 = 1; PN1 = X; Z = X + X; M = 2;
  ENTER RECURSE END;
CHEBY = PN1; RETURN;
SUBROUTINE RECURSE;
BEGIN FOR M = (M,1,N);
BEGIN PN = Z.PN1 - PN2; PN2 = PN1; PN1 = PN END;
  RETURN END END CHEBYSHEV
```

FUNCTIONS

Another sort of subprogram is that resulting from the FUNCTION declaration. The reader should keep in mind that the FUNCTION declaration is only one of several ways in which functions are made available to the program being compiled.

The FUNCTION Declaration

The FUNCTION declaration serves to define those functions of a particularly simple and common kind, which may be expressed by means of a single expression. *A function must be declared before it can be employed in an expression.* Functions may be declared inside of a subroutine, but the result will be the same as if they were declared outside of the subroutine.

GENERAL FORM:

FUNCTION $\mathcal{F}$ $(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n) = \mathcal{E}$

where $\mathcal{F}$ is an identifier which is to be the name of the function. The parameters $\mathcal{P}_1$ through $\mathcal{P}_n$ are identifiers of variables which serve as the parameters of the function, and $\mathcal{E}$ is the expression which defines the function.

Any well-formed expression $\mathcal{E}$ involving the identifiers $\mathcal{P}_i$ and any other identifiers appearing in the program may be used. Identifiers used as parameters of a FUNCTION declaration are independent of identifiers used elsewhere in the program, *even though the identifiers used as parameters are spelled in exactly the same way as the identifiers used outside of the* FUNCTION *declaration.*

(This is subject to the provision that there is no conflict between the declarations of type for the identifiers used in the main program and the desired type for those identifiers, spelled in the same way, which are used as parameters.) All other identifiers appearing in $\mathcal{E}$ that are not included in the parameter list are treated as if they were part of the main program.

The types (integer, floating, Boolean) of the parameters and the type of the value of the function itself are determined by the declarations of type in the same manner as are other identifiers.

EXAMPLES:

```
FUNCTION ROOT (A,B,C) = (-B + SQRT(B*2 - 4A.C))/2A
FUNCTION NORM (X,Y) = SQRT( (U.X*2 + V.Y*2)/(U + V))
FUNCTION NEGEXP(Z) = (1 + Z(0.2507213 + Z(0.0292732
  + 0.0038278Z)))*-4
FUNCTION ARCSINH(S) = LOG(S + SQRT(S*2 + 1))
FUNCTION STROKE(P,Q) = NOT (P AND Q)
```

Evaluation of a Declared Function

The evaluation of a declared function is initiated through a *function call.*

GENERAL FORM:

$\mathcal{G}$ $(\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_n)$

where the identifier $\mathcal{G}$ is the name of the function, and $\mathcal{E}_1$ through $\mathcal{E}_n$ are the expressions representing arguments supplied to the function.

The arguments may be constants, or variables used in the main program, or both. However, it is the responsibility of the programmer to ensure that the expressions representing the arguments of the function agree in type, number, and order with the parameters defined in the FUNCTION declaration. When a function call is encountered in the program, the expressions in the argument list are evaluated and their respective values assigned to the corresponding variables in the parameter list. These current values are those used in the evaluation of the expression specified in the FUNCTION declaration.

EXAMPLES:

```
ROOT(5, C/2, C)
NORM(A,B)
NEGEXP(E)
ARCSINH(11013.2)
STROKE(A OR B, A EQIV B)
```

INTRINSIC FUNCTIONS

There is a small group of functions called *intrinsic functions*, the definitions of which are a part of the compiler and consequently require no FUNCTION declaration. Each of the intrinsic functions is discussed in turn and a tabular summary of them is given below.

MOD ($\mathcal{E}_1$, $\mathcal{E}_2$)

The function MOD requires two integer arguments. The value of the function is the integer obtained as the remainder when the first argument is divided by the second.

MAX ($\mathcal{E}_1$, ⋯, $\mathcal{E}_q$) and MIN ($\mathcal{E}_1$, ⋯, $\mathcal{E}_q$)

The functions MAX and MIN must have two or more arguments. The value of the function MAX will be the value of the largest of its arguments (algebraically); the value of the function MIN will be the value of the smallest of its arguments (algebraically).

The arguments may be either integer or floating-point expressions. If all of the arguments are integers, then the value of the function will be an integer. If any of the arguments is floating-point, then the result will also be floating-point.

SIGN ($\mathcal{E}$)

The function SIGN has a single argument. If this argument is positive, the result will be +1; if zero, the result is also zero; if negative, the result is −1. The result of the SIGN function will be of the same type as that of its argument.

ABS ($\mathcal{E}$)

The function ABS has a single argument. The result of the ABS function will be the absolute value of the argument and will be of the same type as its argument.

PCS ($\mathcal{E}$)

The function PCS is used for interrogating the PROGRAM CONTROL SWITCHES. The value of this function is Boolean in type and is ***true*** if the indicated PROGRAM CONTROL SWITCH is ON and ***false*** if OFF. The units digit

INTRINSIC FUNCTIONS

Name and Description	Type of Function	Type of Argument(s)	Examples
MOD $(X_1, X_2) = X_1 \bmod X_2$	Integer	Integer	X_1 = 100, X_2 = 7, MOD = 2 X_1 = −100, X_2 = 7, MOD = −2 X_1 = 100, X_2 = −7, MOD = 2 X_1 = −100, X_2 = −7, MOD = −2
MAX $(X_1, \ldots, X_k)$, $k \geq 2$	Same as arguments	Integer or floating-point	$A = 1$; $B = 14$; $C = 6$. Y = MAX (A, B, C) $Y = 14$.
MIN $(X_1, \ldots, X_k)$, $k \geq 2$	Same as arguments	Integer or floating-point	$A = 0.1$; $B = 14.0$; $C = 6.1$. Y = MIN (A, B, C) $Y = 0.1$.
SIGN $(X) = \begin{cases} 1, X > 0 \\ 0, X = 0 \\ -1, X < 0 \end{cases}$	Same as argument	Integer or floating-point	If $X = 34$, SIGN $(X) = +1$. If $X = 0$, SIGN $(X) = 0$. If $X = -15$, SIGN $(X) = -1$.
ABS $(X) = \lvert X \rvert$	Same as argument	Integer or floating-point	If $X = -45.67$, ABS $(X) = 45.67$ If $X = +19$, ABS $(X) = 19$.
PCS (N)	Boolean	Integer or floating-point	If PROGRAM CONTROL SWITCH 3 is ON, PCS(3) = 1. If OFF, PCS(3) = 0.

of the argument of the PCS function indicates which of the PROGRAM CONTROL SWITCHES, 0 through 9, is to be interrogated. The argument may be either integral or floating-point. If a MONITOR, TRACE, or DUMP declaration is included in the program, the use of the following control switches may be restricted: PCS 7, PCS 8, PCS 9, and PCS 0. (See CHAPTER X.)

PROCEDURES

Procedures represent the third category of subprograms. A *procedure* is a closed independent routine which may be executed as a subprogram. This independence makes procedures extremely important features of this compiler. A procedure may be written and checked out independently, a collection of these procedures then being retained as a repository of computing techniques. The flexibility built into the argument structure of procedures allows a specific procedure to be tailored to a variety of situations.

There are three means by which a procedure may be made available to a compiled program:

First, the procedure may be taken from the library of machine-language programs which define procedures. These are called *library procedures* (see APPENDIX G).

Second, a machine-language program deck defining a procedure may be included with the cards containing the symbolic program. These are called *external procedures* (see APPENDIX F).

Third, a procedure may be declared in the symbolic language.

The PROCEDURE Declaration

Procedures may be made available to the compiled program by means of the PROCEDURE declaration.

GENERAL FORM:

PROCEDURE $\mathcal{P}$ ($\sim$ $\mathcal{LP}$ $\sim$);
  BEGIN $\mathcal{S}_1$; $\mathcal{S}_2$; $\sim$; $\mathcal{S}_n$; END

where $\mathcal{P}$ is an identifier which names the procedure being declared; the $\mathcal{S}$'s are the statements and declarations making up the definition of the procedure; and $\mathcal{LP}$ is the list of parameters to be used by the procedure. As an alternative form, END may be followed by the name of the procedure and then by a pair of parentheses.

The List of Parameters

The *list of parameters* of the PROCEDURE declaration consists of identifiers and punctuation marks, these identifiers serving as names of the various parameters. An input parameter may be a simple variable or expression. A parameter representing an n-dimensional array must be denoted by an identifier followed by a pair of parentheses which encloses $n-1$ commas. Output parameters have the same form as input parameters, and represent output variables or output arrays. Program-reference parameters may be statement labels—or identifiers representing labels of subroutines and segments, as well as input, output, and format labels. Identifiers in the program-reference parameter list which represent names of functions or procedures must be followed by a pair of parentheses. *A procedure may not call itself.*

The list of parameters of a PROCEDURE declaration may be grouped into three categories. Let $\mathcal{IP}$, $\mathcal{OP}$, and $\mathcal{PRP}$ represent input, output, and program-reference parameters, respectively.

Since any but not all of these three categories may be missing, the list of parameters may assume the following forms:

First: ($\mathcal{IP}$; $\mathcal{OP}$; $\mathcal{PRP}$)
Second: ($\mathcal{IP}$)
Third: ($\mathcal{IP}$; $\mathcal{OP}$)
Fourth: ($\mathcal{IP}$; ; $\mathcal{PRP}$)
Fifth: (; $\mathcal{OP}$)
Sixth: (; $\mathcal{OP}$; $\mathcal{PRP}$)
Seventh: (; ; $\mathcal{PRP}$)

Independence of Declared Procedures

The compound statement defining a procedure is written in terms of the identifiers appearing in the list of parameters of the PROCEDURE declaration, together with any other identifiers required.

The definition of a procedure should be considered as a symbolic program which is independent of the program in which the declaration occurs; that is, all identifiers appearing within a PROCEDURE declaration are defined only in terms of the declaration itself. Identifiers spelled identically both inside and outside of any particular PROCEDURE declaration are in no way associated. There is one exception to this rule: *After a procedure is declared, the identifier which names it is recognized as such throughout the subsequent program except where it is used as a parameter in a* FUNCTION *or* PROCEDURE *declaration.* Indeed, the declaration of a procedure might be construed as adding another feature to the compiler language, since the name of a procedure is recognized throughout the program just as are the reserved words FOR, GEQ, etc.

Declarations Within Procedures

The procedure definition must contain a sufficient number of declarations to describe the identifiers appearing

either as parameters or in any other form within that definition. Parameters representing arrays, functions, or other procedures are identified as such by the punctuation associated with them in the list of parameters. For example, if W(, ,) appears as an input or output parameter, W should not appear in an ARRAY declaration within the PROCEDURE declaration. However, any parameters which represent quantities within the PROCEDURE declaration must have their types specified, either explicitly within a type declaration, or by default. (See CHAPTER V.) These declarations have no force outside the PROCEDURE declaration. A parameter representing a label of one sort or another is identified as such by its inclusion in the program-reference portion of the list of parameters without a trailing pair of parentheses, this constituting a sufficient identification for labels.

Parameters of Value and Name

It is necessary to distinguish two classes of parameters, *parameters of value* and *parameters of name.*

Parameters of value are variables within the procedure. These variables will be set to the values of their corresponding arguments whenever the procedure is called. Any change in them occurring within the procedure (for example, appearing as the left-hand member of an assignment statement) will have no effect on the variable or variables which make up the corresponding argument.

On the other hand, *parameters of name* are associated with their corresponding arguments in all respects. For example, if PAR is a parameter of name for a certain PROCEDURE declaration and, for some call of that procedure, ARG is the corresponding argument, then the effect is exactly as though the identifier ARG were substituted for the identifier PAR throughout the PROCEDURE declaration.

Input variables (or expressions) are parameters of value. Input arrays, all output parameters, and all program-reference parameters are parameters of name. (There is thus no real distinction between an array indicated as an input or as an output parameter.)

Construction of Procedures

As mentioned earlier, a procedure is an independent program complete with its own declarations and statements. This program is contained within the BEGIN ... END pair noted in the general form. The program defining the procedure is entered at the first statement following the BEGIN and continues in accordance with the sequence specified. As with the SUBROUTINE declaration, a RETURN statement must be included at each exit point of the procedure to return control to the point directly following the procedure call.

A procedure must be declared prior to its first use. It may then be referred to from subsequent parts of the program, as well as from within other PROCEDURE declarations. FUNCTION and SUBROUTINE declarations may appear within a PROCEDURE declaration. *However, one* PROCEDURE *declaration may not appear within another procedure.*

If a SUBROUTINE declaration appears within a PROCEDURE declaration, a RETURN statement within the subroutine causes an exit from the subroutine, not from the procedure.

Examples of PROCEDURE Declarations

As a first example, we shall construct a procedure to perform linear interpolation of a tabular function of one variable, V. The parameters of the procedure will be two vectors, X ( ) and Y ( ), representing the independent and dependent variables, a value V of the independent variable, an integer N representing the number of entries in the table, and finally a statement label, RANGE, to which transfer is made in the event that $V < X(1)$ or $V \geq X(N)$. This procedure is to be used as a function, the value of which is the result of the interpolation.

EXAMPLES:

```
PROCEDURE INTERP (X( ), Y( ), V, N; ; RANGE); BEGIN
  INTEGER I, N;
  IF (V LSS X(1) ) OR (V GTR X(N) ); GO TO RANGE;
  I = 1; UNTIL V LEQ X(I); I = I + 1;
  INTERP( ) = Y(I – 1) + (Y(I – 1) – Y(I) ) (V – X(I – 1))/
  (X(I – 1) – X(I) ); RETURN END INTERP( )
```

A procedure for multiplication of square matrices:

```
PROCEDURE MATRIMULT (N, A(,), B(,); C(,) ); BEGIN
  INTEGER I,J,K,N;
  FOR I = (1,1,N); FOR J = (1,1,N); BEGIN
    S = 0; FOR K = (1,1,N); S = S + A(I,K).B(K,J);
    C(I,J) = S END; RETURN END
```

(Note, in calling this procedure, that the matrix C must be different from A and B.)

The following procedure solves a set of n equations in n unknowns.

```
COMMENT SOLVE EQUATIONS WITH SELECTION
  OF BEST PIVOTAL ROW;
PROCEDURE JORDAN (N,A (,); X( ));
  BEGIN INTEGER I,J,K,L,N;
  FOR K = (N + 1, –1, 1); BEGIN D = 0; L = 1;
    FOR I = (2,1,K); IF ABS (A(I – 1,1) ) GTR D;
      BEGIN L = I – 1; D = ABS (A(L, 1) ) END;
    IF L – 1 NEQ 0;
  FOR J = (1,1,K); BEGIN D = A(L,J);
    A(L,J) = A(1,J); A(1,J) = D END;
  FOR I = (1,1,N); X(I) = A(I,1);
```

```
FOR J = (2,1,K); BEGIN D = A(1,J)/X(1);
  FOR I = (2,1,N); A(I – 1, J – 1) = A(I,J) – X(I).D;
  A(N,J – 1) = D END END; RETURN END JORDAN( )
```

The Procedure-Call Statement

A *procedure-call statement* constitutes a complete statement in itself. The entry to a procedure is specified by a procedure-call statement, and may assume any of the following forms:

GENERAL FORMS:

First form: $\mathcal{P}$ ($\mathcal{IA}$; $\mathcal{OA}$; $\mathcal{PRA}$)

where $\mathcal{P}$ is the name of the procedure being called;
$\mathcal{IA}$ is the list of input arguments;
$\mathcal{OA}$ is the list of output arguments; and
$\mathcal{PRA}$ is the list of program-reference arguments.

Any but not all of the argument lists may be missing, resulting in the following alternative forms:

Second form: $\mathcal{P}$ ($\mathcal{IA}$)
Third form: $\mathcal{P}$ ($\mathcal{IA}$; $\mathcal{OA}$)
Fourth form: $\mathcal{P}$ ($\mathcal{IA}$; $\mathcal{PRA}$)
Fifth form: $\mathcal{P}$ (; $\mathcal{OA}$)
Sixth form: $\mathcal{P}$ (; $\mathcal{OA}$; $\mathcal{PRA}$)
Seventh form: $\mathcal{P}$ (; ; $\mathcal{PRA}$)

Arguments of Procedures

Whenever a procedure is called, a list of arguments, or actual parameters, is specified. These arguments may be grouped into three categories: Input arguments, output arguments, and program-reference arguments. Some of these categories may be missing, depending of course upon the specific procedure used.

An input argument may be a simple variable, an expression, or an argument array (see CHAPTER II, *Subscripted Variables*).

An output argument may be a simple variable, a subscripted variable, or an array. A program-reference argument may be a statement label, subroutine label, segment label, input label, output label, or a format label. The name of a function defined by a FUNCTION declaration, or the name of any other procedure, may be used as a program-reference argument when it is followed by a pair of parentheses.

It is important to distinguish between an array and an element of an array. A subscripted variable is an element of an array—it represents a single quantity. An array, however, represents a collection of quantities. When an array is indicated as an argument of a procedure, the procedure is concerned with this entire collection of quantities.

Suppose that the two-dimensional array named M is to be an argument. The notation used for this argument is M(,), the two empty subscript positions indicating that M is two-dimensional. It is also possible to specify that a portion of some array be given to a procedure as an argument. For example, if a procedure requires that a certain argument be a one-dimensional array, the array could be chosen as the (I + 1)th row of M (,) by writing M(I + 1,). The single empty subscript position indicates a one-dimensional array. In similar fashion, the (L – 2K)th column of M would be written as M(,L – 2K). In general, the name of an array followed by a subscript list which contains empty subscript positions specifies an array with dimensions equal to the number of empty subscript positions. This holds whether or not some of the subscripts are specified.

To specify a program-reference argument which is to be a label, or the label of a subroutine, input-data list, output-data list, or format-data list, it is necessary only to write the desired identifier. A function or procedure is specified by writing the name of the function or procedure followed by a pair of parentheses, for example HAVERSINE( ).

Note that we now have a distinction between a function and an evaluated function. An evaluated function has its arguments specified; it represents the quantity obtained by applying the definition of the function to those arguments, and is thus an expression. A function, however, represents only the definition.

Examples of Procedure-Call Statements

As an example of a procedure-call statement, suppose that a procedure called INVERT has been defined to evaluate the inverse and also to calculate the determinant of a given matrix. The arguments of this procedure are to be:

Input:	The order of and the name of the matrix, the determinant and inverse of which are to be evaluated;
Output:	The array which is to receive the inverse and the variable which is to receive the computed value of the determinant;
Program-reference:	A label of a statement to which transfer is to be made if the matrix is singular.

The user would then write:

```
INVERT (N, A( , ); B( , ), D; ERROR4)
```

to set D to the value of the determinant of the N × N matrix A(,) and to set B(,) equal to the inverse of the matrix A(,). A transfer to the statement labeled ERROR4 will occur if A(,) is singular.

As a second example, suppose that a vector of data points is to be treated by a least-squares smoothing process. Suppose a procedure called SMOOTH is available. The programmer might write:

```
SMOOTH ( K, X( ); Y( ) )
```

where the input parameters are first, the number of data points K, and second, the name X( ) of the vector containing them, while the output parameter Y( ) is the name of the vector to receive the computed results.

Functions Used As Arguments

Any function—library, external, or declared—may be used directly as a program-reference argument, with the exception of the intrinsic functions, listed on page 7-3. The exclusion of these intrinsic functions as program-reference arguments was intentional, since the compiler cannot treat this case directly. If a procedure requires a function as one of its arguments, and the user desires to specify it as an intrinsic function, that intrinsic function may be renamed by the use of the FUNCTION declaration. For example, if ABS( ) is to be the function specified, write

```
FUNCTION F(X) = ABS(X)
```

and give the procedure F ( ) for the argument.

No provision has been made in the compiler for using a function the arguments of which have been in part specified and in part left empty. This situation causes little inconvenience, however, since the FUNCTION declaration may be used in lieu of such a feature. For example, assume that a function of two arguments Q(X, Y) is available and that it is desired to specify, as an argument to a procedure, that function of one argument Y which is defined by always setting X to (A − B)*3. If the declaration

```
FUNCTION QPRIME(Y) = Q((A – B)*3,Y)
```

is included in the symbolic program, then using the argument QPRIME( ) will produce the desired results.

Functions Defined by Procedures

Some procedures define functions, i.e., the procedure with a given set of arguments represents a quantity. This is the extended form of the evaluated function mentioned in CHAPTER II.

A procedure which is to serve as a function must include a *procedure-assignment statement.*

GENERAL FORM:

$\mathcal{P}$ ( ) = $\mathcal{E}$

where $\mathcal{P}$ is the name of the procedure being declared and $\mathcal{E}$ is an expression. The effect of this statement is to assign the value of $\mathcal{E}$ to the procedure. Immediately after this statement is executed, a RETURN statement must be executed. As an example of this sort of construction, consider the integration procedure using Simpson's Rule:

```
PROCEDURE SIMPS (A,B,EPSILON; ; F( ) ); BEGIN
K = L = F(A) + F(B); H = B – A;
GO TO ITER; UNTIL ABS( (K – 2M)/K) LSS EPSILON;
BEGIN ITER..Q = H/2; S = 0; M = K;
FOR X = (A + Q,H,B) ; S = S + F(X);
K = L + 4S; L = L + 2S; H = Q END;
SIMPS( ) = K.Q/3; RETURN END SIMPS( )
```

In the above definition of SIMPS ( ), A and B are the lower and upper limits of integration, respectively. EPSILON is the maximum tolerable error in the result, and F ( ) is the function to be integrated.

The single value associated with the procedure SIMPS ( ) is the value of the definite integral indicated. The type of the procedure is determined from the declarations of type within the PROCEDURE declaration. Suppose that we wish to evaluate the equation:

$$J = 4\left[\sqrt{\left(\int_{X-Y}^{X+Y} G_3(T)dT\right)^3}\right]$$

The assignment statement

```
J = 4SQRT (SIMPS ( X – Y, X + Y, 1**–6; ; G3 ( ) )*3 )
```

would suffice. Note that the call on the procedure representing a function constitutes an expression which may be combined with other expressions to form a statement.

EXTERNAL PROGRAM DECLARATIONS

By the use of this kind of declaration, a programmer may define a statement or a procedure in terms of machine language and include it in a compiled program.

GENERAL FORMS:

First form:

```
EXTERNAL STATEMENT £
```

where £ is the label of an external statement.

The first form declares the program represented by £ to be a statement which will behave similarly to any other active statement in the language. The label of this statement will be £.

If the machine-language program defining the external statement is to be referred to from more than one point in the program, a subroutine may be used to enclose the

declaration EXTERNAL STATEMENT £. All references to the external statement £ will then be made by a reference to that subroutine. The declaration is treated as a statement which initiates the execution of the program £.

Second form:

EXTERNAL PROCEDURE £($\mathcal{P}_1$, ⋯, $\mathcal{P}_n$); { INTEGER / BOOLEAN / FLOATING / REAL } £

where £ is the label of the external program, and $\mathcal{P}_1$, ⋯, $\mathcal{P}_n$ is a list of parameters.

The second form defines the program represented by £ to be a procedure which will behave like any other procedure in the language. The declaration of the second form above must be made prior to a call on that procedure. The call on an external procedure is precisely analogous to that of any defined procedure discussed earlier.

Note that if the procedure is to define a function, then its declaration of type must follow it immediately; if not, the declaration of type may be omitted.

Both forms indicate to the compiler that a machine-language program defining £ follows the FINISH card of the symbolic program (see APPENDIX F).

EXAMPLES:

First form:

```
EITHER IF V GTR NMAX; BEGIN EXTERNAL STATEMENT
ERROR; GO RESET END; OR IF K LSS EPS; GO ERROR END
```

Second form:

```
EXTERNAL PROCEDURE COMPLEXMULT (A, B, C, D; X, Y);
FLOATING COMPLEXMULT; SREAL = 0; SIMAG = 0;
FOR I = (1,1,10); BEGIN COMPLEXMULT (AREAL(I),
AIMAG(I), BREAL(I), BIMAG(I); TREAL, TIMAG);
SREAL = SREAL + TREAL; SIMAG = SIMAG + TIMAG END
EXTERNAL PROCEDURE SPERR(;;FRDEC);
IF V GTR NMAX; SPERR(;;NERR)

PROCEDURE MATRIMULT (M, N, P, A( ), B( ), ROW, COLUMN,
OUTTAPE; C( )); BEGIN INTEGER OTHERWISE;
FLOATING A, B, C; EXTERNAL PROCEDURE REWIND (U);
EXTERNAL PROCEDURE MAGREAD (N, U, A( ));
EXTERNAL PROCEDURE MAGWRITE (N, U, A( ));
REWIND (ROW); REWIND (COLUMN); REWIND (OUTTAPE);
FOR I = (1, 1, M); BEGIN MAGREAD (P, ROW, A( ));
FOR J = (1, 1,N); BEGIN MAGREAD (P, COLUMN, B( ));
BEGIN S = 0; FOR K = (1, 1, P); S = S + A(K) B(K);
C (J) = S; REWIND (COLUMN) END;
MAGWRITE (N, OUTTAPE, C ( )) END; RETURN END
```

Machine-Language Procedures

Library procedures and all external programs are written in the machine language of the BURROUGHS 220. Any features of the compiler—for example, input-output procedures, multiple-precision and partial-word arithmetic, detection of overflow, etc.—may be made available to the compiled program through machine-language procedures. The standard library contains a selection of commonly used procedures which evaluate the elementary functions and perform certain operations, such as the printing of error messages, or editing of output information according to FORMAT declarations. While many of these functions may be expressed in the compiler language, the convenience of having these procedures 'on call' without the programmer directly providing their definitions makes their inclusion in the library worthwhile.

APPENDIX F describes the preparation of library and external procedures; APPENDIX G describes the library procedures currently available.

INPUT
PREPARATION OF DATA CARDS
PREPARATION OF PAPER TAPE
OUTPUT
FORMAT
EDITING

VIII . . .

input-output techniques

THIS CHAPTER DISCUSSES those features of the compiler relating to communication of information between the computer and the input-output equipment. In general, the compiler input-output operations are accomplished by means of machine-language procedures (either external procedures or library procedures). Three declarations are provided to aid in the input-output processes—the INPUT, OUTPUT, and FORMAT declarations.

INPUT OF INFORMATION

The INPUT Declaration

The INPUT declaration associates with identifiers ordered sets of variables, values of which are to be read into the computer as units. These sets of variables are called *input-data lists* and the identifiers are termed *input labels*. The input label with its associated input-data list is termed an *input-list element.*

GENERAL FORM:

INPUT ($\mathfrak{g}(\mathfrak{DL})$, ⋯, $\mathfrak{g}(\mathfrak{DL})$)

where each $\mathfrak{g}$ is the identifier declared to be the label of the corresponding input-data list $\mathfrak{DL}$. As explained in CHAPTER V, the parentheses around the input list in such a declaration are included at the programmer's option. Input-data lists consist of *input data-list elements* separated by commas. The simplest of these elements is merely a list of variables.

First form:

$\mathfrak{V}_1$, $\mathfrak{V}_2$, ⋯, $\mathfrak{V}_n$

where each $\mathfrak{V}$ may be either a simple variable or a subscripted variable.

EXAMPLE:

```
INPUT DATA1 (X,Y,Z), DATA2 (P(I), Q(I))
```

This declaration defines DATA1 to be the set of variables X, Y, and Z (in that order) and DATA2 to be the set P(I) and Q(I), using the current value of I for the subscript.

The next list element to be considered is termed the *iterated variable.*

Second form:

FOR $\mathfrak{V}_1 = \mathfrak{gL}$; ⋯; FOR $\mathfrak{V}_k = \mathfrak{gL}$; $\mathfrak{V}$

where FOR $\mathfrak{V}_1 = \mathfrak{gL}$ through FOR $\mathfrak{V}_k = \mathfrak{gL}$ are FOR clauses which control the iteration of the variable $\mathfrak{V}$. One or more of these FOR clauses may precede $\mathfrak{V}$.

EXAMPLES:

```
INPUT ( VECTOR(FOR L = (1,1,F); Z(L) ),
MATRIX ( FOR I = (1,1,N); FOR J = (1,1,M); E(I,J) ) )
```

This declaration defines VECTOR to be the label of the input-data list consisting of the variables

Z(1), Z(2), Z(3), ⋯, Z(F − 1), Z(F),

and MATRIX to be the label of the input-data list consisting of the variables

E(1,1), E(1,2) ⋯, E(1,M), E(2,1), ⋯, E(N,M).

Note that the values for F, M, and N must have been assigned elsewhere in the program.

The remaining forms of the input-data list consist of combinations of the first and second forms. In the first form, any $\mathfrak{V}$ may be replaced by any input data-list element and still be a valid input-data list. In the second form, the $\mathfrak{V}$ may be replaced by any input data-list element enclosed in parentheses and still be a valid input-data list. In this case, everything in the enclosed input-data list is iterated.

Third form:

$\mathfrak{DL}$, $\mathfrak{DL}$, $\sim$, $\mathfrak{DL}$

Fourth form:

FOR $\mathfrak{V}_1 = \mathfrak{gL}$; $\sim$; FOR $\mathfrak{V}_k = \mathfrak{gL}$; ($\mathfrak{DL}$)

The input-data lists given by the third and fourth forms may of course be used as input-data lists within their own definitions, thus providing for a very high degree of flexibility.

EXAMPLE:

```
INPUT EQUATIONS (N, FOR I = (1,1,N);
  (FOR J = (1,1,N); M(I,J), C(I) ) )
```

This input-data list consists of the variables N, M(1,1), M(1,2), ..., M(1,N), C(1), M(2,1), ..., M(2,N), C(2), M(3,1), ..., M(N,N), and C(N).

Note that as soon as a variable (in this case N) has been read into the computer, it is immediately available for use in a FOR clause or within a subscript expression.

Input Procedures

Machine-language procedures are employed to obtain numbers from the input medium. The label of an input-data list usually will appear as a program-reference parameter of an input procedure. For convenience the frequently used READ procedure is included in the library and will be discussed below.

There is an important feature of all input procedures which should be noted, so that caution may be exercised in their use. *Any* FOR *clause with an* INPUT *declaration affects the value of the variable being stepped in exactly the same manner as a* FOR *clause controlling a statement.* Thus if an input procedure is contained within the scope of a FOR clause and the INPUT declaration to which it refers contains a FOR clause, *the variables stepped by the two iterations should be different.* This also applies to the output procedures described later in this chapter.

The READ Procedure

The READ procedure provides data input for compiled programs from punched cards, magnetic tape, and paper tape. It may be called in either of two ways:

GENERAL FORMS:

First form:

READ (; ; $\mathfrak{gDL}$)

Second form:

READ (; $\mathfrak{s}$; $\mathfrak{gDL}$)

where $\mathfrak{gDL}$ represents the label of some input-data list and $\mathfrak{s}$ is a Boolean variable.

This procedure obtains information from the input buffer and assigns values to the variables in accordance with the specifications of $\mathfrak{gDL}$. The buffer is filled a number of times sufficient to supply all the values requested by $\mathfrak{gDL}$; if the entries in the last buffer load are not requested, they are lost. If the second form was used, the Boolean variable $\mathfrak{s}$ is set to ***true*** when a sentinel card is found and the input process is terminated (see *Preparation of Data Cards*, below); $\mathfrak{s}$ is otherwise set to ***false***. Using the first form will cause the word SENTINEL to be ignored.

The REED Procedure

The REED procedure is used to load the input buffer whenever more information is requested by its equivalence, READ. The word REED is not a reserved word in the compiler language, and the procedure cannot be referred to explicitly in a program.

The REED procedure included in the library of the standard version of the compiler allows only CARDATRON input. To facilitate the reading of data from paper tape, a compiler containing the proper REED must be used. Since a compiler has access to only one REED procedure, data from cards, paper tape, or magnetic tape cannot be read simultaneously. However, the same program will accept input from any *one* of these sources, depending upon the characteristics of the compiler used (see APPENDIX A).

Preparation of Data Cards

Data cards are punched with a digit five in column 1; the remainder of the card is available for data. An integer is punched as a string of no more than ten contiguous digits, preceded by an optional + or −. Floating-point numbers are punched as a string of digits containing a decimal point.†

No more than eight significant digits may appear in a floating-point number. Leading zeros are not significant; trailing zeros are. A sign may precede the number. A scale factor (power of ten) may be attached to a floating-point number by following the number with a ',' (not '**'), an optional sign, and a two-digit integer.

At least one blank column must separate the numbers on a card. A number may not be broken between successive cards.

† As opposed to the representation used for constants in the symbolic program, the decimal point may also appear on data cards at either end of the string.

Alphanumeric information may be stored in the computer in the form of integer quantities. This is accomplished by punching on a card the alphanumeric input information bracketed with semicolons; consequently, the character ';' may not be used within the alphanumeric string itself. For example, if 30 consecutive characters are to be entered in columns 13 through 41 of a card (six words), then semicolons must be punched in both column 12 and column 42.

The string of alphanumeric characters may extend over more than one card. Column 2 of the next card will be considered a continuation of column 80 of the last card read. The string is terminated by a semicolon. However, if the INPUT declaration specified in the READ procedure is satisfied before the semicolon is reached, the remaining information on the current data card will be lost.

Arrays of variables reserved for the storage of such data should be declared of type INTEGER. One integer, ten digits in length, is stored in the computer for every five consecutive characters of alphanumeric information. The internal representation is two digits per alphanumeric character, as described in APPENDIX C of *Operational Characteristics of the* BURROUGHS *220 Electronic Data Processing System* (Bulletin No. 5020A). If the number of columns of alphanumeric information is not a multiple of five, the last integer formed will have numeric zeros (alphanumeric blanks) appended. The integer digits representing the residual characters will be justified left in this word.

An asterisk will cause all information to its right on the card to be ignored (if the asterisk is not within an alphanumeric string).

EXAMPLES:

1 37 –4724 +0394	are integers
3.0 –.9 +3.1416 8.	are floating-point numbers
–2.99,9 32.4,–13 +7.2,+2	are floating-point numbers with scale factor

; THE GOAT OF HOGAN ; is an alphanumeric entry

A sentinel card is identified by punching the word SENTINEL starting at column 2 and followed by a blank column:

COLUMNS	ENTRY
1	5
2-9	SENTINEL
10	(blank)
11-80	(*optional*)

Preparation of Data for Paper Tape

The general rules associated with the preparation of data cards also apply to the punching of data on paper tape. Data on paper tape is read in blocks of 16 words, each block representing an 80-column alphanumeric card image. The first two digit positions of the first word ($sL=22$) of each data block are not scanned for information. This permits convenient card-to-tape conversion, whenever needed. Both the single-frame code and the two-decimal-digit B 220 code may be used. All special-character codes are tested and, if needed, translated into B 220 internal representation during input.

The *sentinel block* is formed by punching the word SENTINEL in the first two words as follows:

WORD	SINGLE-FRAME CODE	B 220 CODE
1	2 SENT	0 0062455563
2	2 INEL	0 4955455300
3-16	(*optional*)	

Note that the techniques explained here are applicable only with the paper-tape REED procedure supplied by BURROUGHS upon request. If this does not meet the particular needs of the application, a suitable REED can be written by the user (see APPENDIX F).

OUTPUT OF INFORMATION

The OUTPUT Declaration

The OUTPUT declaration associates with identifiers the ordered sets of expressions which are to be written out of the computer as groups. These sets of expressions are called *output-data lists* and the identifiers are termed *output labels.*

GENERAL FORM:

OUTPUT ($\mathcal{I}$ ($\mathfrak{DL}$), ..., $\mathcal{I}$ ($\mathfrak{DL}$))

where each $\mathcal{I}$ is the identifier declared to be the label of the corresponding output-data list $\mathfrak{DL}$. (See CHAPTER V for the explanation of the use of the outside parentheses.) Output-data lists are constructed in a manner precisely analogous to input-data lists *with the following exception: Although only values of variables may be read into the computer, the value of any expression may be written out. Thus, wherever a variable has been specified in the description of an input-data list, the reader may substitute an expression in an output-data list.*

EXAMPLE:

OUTPUT RESULTS ((R – 1) (S – 1), N(R), N(S), 8.4)

Output Procedures

Machine-language procedures are employed to transmit numbers to the output medium. The label of an output-data list appears as the program-reference parameter of an output procedure.

The WRITE Procedure

The WRITE procedure provides for the edited output of a compiled program on the LINE PRINTER, HIGH-SPEED PRINTER, the CARD PUNCH, or the SUPERVISORY PRINTER.

First form:

WRITE (; ; $\mathcal{G}\mathfrak{D}\mathfrak{L}$, $\mathcal{G}\mathfrak{F}\mathfrak{L}$)

Second form:

WRITE (; ; $\mathcal{G}\mathfrak{F}\mathfrak{L}$)

where $\mathcal{G}\mathfrak{D}\mathfrak{L}$ is the label of an output-data list and $\mathcal{G}\mathfrak{F}\mathfrak{L}$ is the label of a *format-data list.*

The effect of the first form is to edit the output and load the output buffer with the values specified by the output-data list $\mathfrak{D}\mathfrak{L}$, in accordance with the format specified by $\mathfrak{F}\mathfrak{L}$. The second form is used to fill the output buffer with the messages indicated by the format-data list $\mathfrak{F}\mathfrak{L}$. The type of output medium used depends on the particular RITE procedure incorporated in the library, and on the type of activation phrase used in the format-data list.

The RITE Procedure

The RITE procedure is frequently used for the transmittal of edited output information to the output medium. (This procedure is not required when the activation phrase is intrinsic to the WRITE procedure itself.)

The word RITE is not a reserved word; RITE is a procedure, and is referred to automatically through the WRITE procedure. The standard library contains a RITE procedure for the LINE PRINTER; however, the user may replace it by any other machine-language program (see APPENDIX F).

CONSTRUCTION OF FORMATS†

The FORMAT Declaration

The FORMAT declaration has been designed specifically for use with the WRITE procedure, and serves to associate labels with format-data lists which describe the appearance of the output page or card. Such a label then is used as a parameter by the WRITE procedure.

GENERAL FORM:

FORMAT ($\mathcal{G}(\mathfrak{F}\mathfrak{L})$, $\mathcal{G}(\mathfrak{F}\mathfrak{L})$)

where the $\mathfrak{F}\mathfrak{L}$'s are format-data lists and the $\mathcal{G}$'s are the associated labels (outside parentheses again are optional).

The format-data list consists of a sequence of format data-list elements which are separated by commas and perhaps grouped by parentheses. These elements occur in two forms:

First form:

rLw.d

Second form:

$\mathfrak{F}\mathfrak{S}$

In the first form, the symbols *r*, *w*, and *d* represent integers and *L* represents a letter. The integers *r* or *d* or both are sometimes omitted, reducing the first form to *Lw.d*, *rLw*, or *Lw* in these cases.

The integer *r* specifies the number of times an element is to be repeated. If *r* is omitted, the element is executed once. Elements of the first form are divided into two classes: *editing elements and activation phrases.*

In the second form, $\mathfrak{F}\mathfrak{S}$ represents any string of characters which does not contain an '*'. This element is used for placing alphanumeric titles or other indicative information in the output line. It is called a *format string.*

Editing Elements

A numeric editing element specifies how a number is to be edited. There are six such elements:

I*w* The I element specifies that an integer is to be printed (or punched) in a field *w* columns wide. The integer will be normalized right in that field and will have its leading zeros suppressed. If the integer being edited is negative, a '–' (minus sign) precedes it. The value of *w* must be sufficiently large to accommodate the largest integer to be encountered together with any possible minus sign.

X*w.d* The X element specifies that a floating-point number is to be truncated to *d* places following

† These formats are not to be confused with the format bands used in the BURROUGHS CARDATRON.

the decimal point and printed in a field w columns wide. The value w must be sufficiently large to accommodate the largest number to be printed along with its decimal point and any possible minus sign.

F$w.d$ The F element specifies that a floating-point number is to be truncated to d significant digits and printed (or punched) in floating-point form as follows:

A '−' (if the number is negative), a '.', d digits, a ',', a '−' (if the power of 10 associated with the number is negative), and two digits representing that power of 10. Thus to accommodate negative numbers $w \geq d + 6$.

S$w.d$ The S element specifies that a floating-point number is to be truncated to d digits and printed (or punched) in a field w columns wide. A decimal point will be inserted at the appropriate position to cause the printing (or punching) of d significant digits if possible. If a number is less than 0.1 in absolute value, zeros will be inserted between the decimal point (which will appear at the extreme left) and any significant digits printed, punched, or typed.

Aw The A element allows integers to be printed (or punched) in their alphanumeric equivalents. A single A phrase will produce, as an output, w characters, five characters of alphanumeric information being translated from each word or ten-digit integer. If w is not a multiple of five, the least significant portion of the last word will be ignored.

Bw The element Bw will cause w blank columns to be inserted in the edited line.

If in any case the field width w which has been specified is less than the width required by the output information, or if there are undefined conditions in the format specifications, an asterisk will be printed in the corresponding field.

EXAMPLES:

We list here several elements and a typical result of the editing they specify. In each example the first of the two lines indicates the result of the editing process, where the symbol # indicates an editing space. The line directly below each example shows the equivalent line as printed.

ELEMENTS	RESULT (continued)
417	#####13#−72431####342#####0
	13 −72431 342 0
3X5.2	##.13#−.52#1.74
	.13 −.52 1.74
X12.10	#.0000000015
	.0000000015
F10.3	##.472,−03
	.472,−03
F12.4	##−.3942, 073### .4311,−03###.0000, 00
	−.3942, 07 .4311,−03 .0000, 00
5S9.5	###3427.1##−32.993##−.10206###13788.###.00014
	3427.1 −32.993 −.10206 13788. .00014

A12. The use of this phrase, in conjunction with
4856596245434845626345590057 41

will result in the printout
HORSECHESTER

Format Strings

The phrase *FS* inserts the characters comprising the string FS into the line being edited.

EXAMPLES:

PIPE DIAMETER
TRANSCONDUCTANCE-MICROMHOS
HIGH - LOW - CLOSE - NET CHG.
DURCHSCHLAGFESTIGKEIT - VOLT

Activation Phrases

An *activation phrase* specifies that the line described by the preceding elements is to be sent to the output device.

Whenever a line is written out in this manner, the image in memory of the line is reset to blanks. Thus, if an activation phrase is repeated, only the first execution produces any printed results; the repeat merely provides vertical spacing. Four activation phrases are provided:

Ww The W phrase provides for output under control of the RITE procedure. Unless the standard RITE procedure is replaced by one that is more suitable to the user's needs, the output will be emitted on the LINE PRINTER. The value of w specifies the 'c-digit' to be used for printer control. If w is omitted, it is assumed to be zero. If the control panel is wired as specified in BURROUGHS CORPORATION TECHNICAL BULLETIN No. 17, *Control Panel Wiring for Type 407 with* BURROUGHS *205 or 220* CARDATRON, w is interpreted as follows:

w	RESULT
0	Single space before printing
1	Eject page after printing
2	Single space before and after printing
3	Eject page before printing
4	Double space before printing
5	Skip to channel 2 before printing
6	Double space before printing, single space after printing
7	Skip to channel 3 before printing

P The P phrase specifies that the edited line is to be punched into a card. The action is initiated from within the WRITE procedure. Note that, with the exception of the Aw element, the edited form of the numbers is compatible with requirements for input-data cards.

The output cards produced by the WRITE procedure, in accordance with the specifications of the appropriate format-data list, may be used for input to the computer, under control of the READ procedure. On all such cards, alphanumeric output must be enclosed in semicolons, which may be inserted through use of the format-string element in the format-data list.

EXAMPLE:

```
*;ALGOL;*
```

Also, the specified format-data list must commence with the format-string element *5* which will be converted into a 5-punch in column 1 for a format select on input.

Cw The action of the C phrase combines those of the W and P phrases. Only the first 80 columns of the edited line will be punched.

Tw The T phrase specifies that the edited line is to be printed on the SUPERVISORY PRINTER. The printout takes place from inside the WRITE procedure with no reference to the RITE routine, which is used only in conjunction with the W phrase. The value of w specifies the number of carriage returns to be executed prior to printing.

Repeat Phrases

A list of phrases may be placed in parentheses to constitute a compound phrase. These parentheses may be nested to any depth.

Definite-Repeat Phrase:

$r(\mathfrak{FL})$

Indefinite-Repeat Phrase:

$(\mathfrak{FL})$

where $\mathfrak{FL}$ is a format-data list. The definite-repeat phrase uses the format-data list r times in succession; the indefinite-repeat phrase uses the format-data list repeatedly until there are no more variables to print.

An entire format-data list is treated as if it were enclosed in parentheses specifying indefinite repeat. The interpretation of a format is terminated when there are no more variables to print and the right parenthesis of an indefinite repeat is encountered. If there are fewer variables to print than are called for, the I, X, F, S, and A phrases are interpreted as blank-insertion phrases, Bw, to fill in the remaining spaces.

EXAMPLES:

```
3(5F15.8,W0)
*TITLE*
*5*,B10,*$*,A26,*$*,P
```

The first example is equivalent to:

```
5F15.8,W0,5F15.8,W0,5F15.8,W0
```

The second example will print a line which reads

```
TITLE
```

The third example will punch a card according to the format:

COLUMNS	CONTENTS
1	5
2-11	blanks
12	$
13-38	alphanumeric information
39	$
40-80	blanks

SEGMENTATION
OVERLAYS

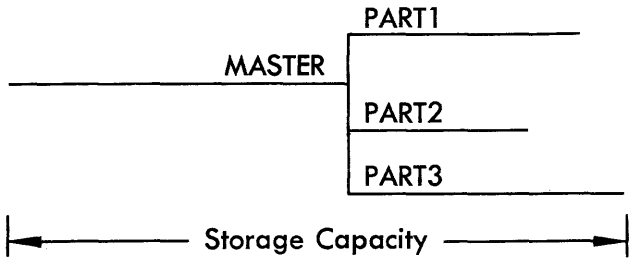
IX . . .

overlay techniques

THE USE OF STORAGE OVERLAYS, though not essential for a majority of problems, is required whenever a program is too large for the amount of available storage. This condition is indicated during compilation by the message MEMORY CAPACITY EXCEEDED.

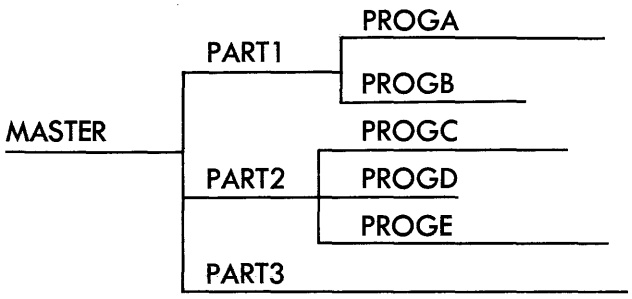
In such a case the program should be coded in segments; these segments are then called into memory as a sequence of overlays under control of a master routine. *Once segmentation is employed in a program, the remainder of that program must consist only of segments.*

To illustrate this graphically:



Here MASTER is the name of the master routine and PART1, PART2, and PART3 are the names of the segments. The lengths of the horizontal lines may be taken as proportional to the number of instructions in the master routine and in the various segments. Note that in the above example the total number of storage locations is only that required by the master together with its longest segment, PART3.

Each of the segments may in turn have subsegments, which may be further subdivided into other subsegments. All of these segments then become *the master routines for control of their respective subsegments.* For example:



The master for PART1, PART2, and PART3 thus is MASTER; the master for PROGC, PROGD, and PROGE is PART2, etc.

THE SEGMENT DECLARATION

The SEGMENT declaration is used to indicate the division of the program into segments.

GENERAL FORM:

SEGMENT $\mathcal{I}$; BEGIN S_1; S_2; ⋯; S_n END

where $\mathcal{I}$ is an identifier which serves as the label of the segment, and S_1 through S_2 are the statements which make up the segment. If desired the segment label may follow the word END as is the case with any compound statement.

For example, the segmentation diagramed above would be written as:

```
MASTER..⋯⋯⋯⋯⋯;
  SEGMENT PART1; BEGIN⋯⋯⋯⋯⋯;
    SEGMENT PROGA; BEGIN⋯⋯END PROGA;
    SEGMENT PROGB; BEGIN⋯⋯END PROGB
      END PART1;
```

```
SEGMENT PART2; BEGIN ~~ ~~ ~~ ~~ ~~ ;
  SEGMENT PROGC; BEGIN ~~ ~~ END PROGC;
  SEGMENT PROGD; BEGIN ~~ ~~ END PROGD;
  SEGMENT PROGE; BEGIN ~~ ~~ END PROGE
    END PART2;

SEGMENT PART3; BEGIN ~~ ~~ END PART3
```

THE OVERLAY STATEMENT

The OVERLAY statement is used to call a segment of the compiled program into storage.

GENERAL FORM:

OVERLAY *g*

where *g* is the segment label of the segment to be called. There are four rules governing the use of the OVERLAY statement:

First: Any segment may be overlaid only by a segment of the same immediate master.

Second: The OVERLAY statement which calls a segment into storage must appear in the master routine corresponding to the segment being called. Otherwise, segments may be called in as many times as the user wishes, in any order.

Third: The OVERLAY statement does not by itself produce a transfer to the segment being called. Such a transfer must be provided for separately by the user. *The segment label may not be used as a statement label for effecting this transfer.*

Fourth: The return from a segment to its immediate master, or to a master several segments removed, must be initiated through some type of transfer statement similar to those used for entry into the segment.

EXAMPLE:

```
BOOLEAN S; REAL A, B; INTEGER OTHERWISE;
ARRAY A(200);
RD..READ(;S;INDAT); IF S; BEGIN OVERLAY SENTINEL;
N = N + 1; GO TO START END; N = N + 1; OVERLAY ZERO;
GO STARTP;
INPUT INDAT (B, C, FOR I = (1, 1, 200); A(I) );
FORMAT FR (I5, 5F20.8, W0);

SEGMENT ZERO; BEGIN STARTP..
FOR I = (1, 10, 200); WRITE (;;OUTP, FR); GO TO RD;
OUTPUT OUTP (I, FOR J = (I, 2, I + 8);
SIN (COS(LOG(A(J))))) END ZERO;

SEGMENT SENTINEL; START..BEGIN FOR I = (1, 10, N - 10);
WRITE (;; FR, OUT); WRITE (;;OUTR, FRR);
STOP 0757007250; GO RD;
OUTPUT OUT(I, FOR J = (I, 2, I + 8); A(J)),
OUTR(I, FOR J = (I, S, I + MOD(N, 10) - 2); A(J));
FORMAT FRR (*I = *, I4, 5(*G(I) = *, F14.6), W0, W3)
  END SENTINEL; FINISH;
```

ERROR MESSAGES

MONITORING

OBJECT PROGRAM LISTING

X . . .

diagnostic facilities

THE COMPILER HAS BEEN EQUIPPED with a set of diagnostic aids. These aids take the form of error messages concerning the symbolic program, error messages indicating that computation has produced an undefined result, and monitor, dump, and trace facilities. A listing of the compiled program may also be obtained.

For purposes of presentation, these diagnostic facilities will be grouped into two categories: those aids operative at the time of compilation, and those operative at the time the object program is run. In general, those in the first category are concerned with the syntax of the symbolic program; those in the second category are useful for the verification of program logic.

AIDS AVAILABLE AT COMPILE TIME

One valuable diagnostic aid is a listing of the symbolic program. This may be obtained during the compilation process. This listing will be given unless PROGRAM CONTROL SWITCH 4 is depressed.

The first step in checking out a program should be the determination of its syntactical correctness. Of considerable importance in helping the programmer to achieve this objective are the error messages provided automatically by the compiler.

Whenever an error is detected, the compiler attempts to continue processing the symbolic program. Since the compilation technique of necessity depends on the syntax of the symbolic program for its classification of identifiers and statements, an error in syntax may well produce a misclassification. If the remainder of the symbolic program is processed with the compiler in such a 'confused' state about the characteristics of the program, a proliferation of error messages may result.

If things get altogether out of hand, the compiler may even come to a disorderly stop. Although such generation of spurious error messages is unfortunate, it has been considered preferable to attempt to continue processing, in order to discover as many errors as possible in a single pass rather than to force the compiler to abandon the compilation after the first error is detected.

The programmer himself must make the distinction between genuine errors and those introduced by previous errors. However, it should not be assumed that all errors in syntax will produce error messages.

No usable compiler can provide error messages which explicitly describe every possible syntactical error. However, sufficient information usually is gained from the error message to permit straightforward correction of the program.

The following is a list of error messages and some suggestions as to their possible cause.

COMPILER CAPACITY EXCEEDED

The internal storage capacity of the compiler has been exhausted. The compiler stops with rC = 0000007777.

To overcome the problem of insufficient symbol storage space, limited corrective measures may be adopted which will result in a more economical use of the associative memory of the compiler:

All identifiers and names of labels should be shortened to five characters or less.

A number of simple variables of the same type may be grouped into a two-dimensional square array, or as nearly square as possible, to conserve symbol table space.

A reorganization of the symbolic program in accordance with conventions discussed in CHAPTER XI will reduce forward references to a minimum, thereby making possible the conservation of additional memory space.

As a last resort, a procedure defined in the ALGOL program may be rewritten in machine language and incorporated as an external procedure. (See APPENDIX F.)

CONSTANT OUT OF RANGE

A constant too large for the internal representation of the BURROUGHS 220 was either written in the symbolic program or was formed when the compiler combined constants arithmetically.

DUPLICATE PROCEDURE NAME

The identifier assigned as the name of a procedure by a PROCEDURE declaration (or the name of a function by a FUNCTION declaration) has appeared in another context in the program.

DUPLICATE LABEL

An identifier has been employed in another context in the program. (Note that this may be the result of calling a function or procedure before it has been declared.)

EXTERNAL PROGRAM NOT DECLARED

The external procedure or external statement declared on some name card heading a machine-language deck was not declared within the symbolic program.

EXTRA OPERAND

This error message may arise from a wide variety of causes. Check for omitted commas, semicolons, or other punctuation; for proper spelling of reserved identifiers, and for a space imbedded within an identifier; or for the omission of a space between contiguous identifiers.

EXTRA LEFT PARENTHESIS

This error message may occur only at the end of processing the symbolic program. It is usually caused by the inclusion of a spurious 'BEGIN' or '(' or by the omission of a required 'END' or ')'. Errors in the syntax of FOR or alternative statements, or in any of the declarations, may also introduce unwanted left parentheses into the program even though the programmer did not explicitly write them.

EXTRA RIGHT PARENTHESIS

This error is usually caused by the inclusion of a spurious 'END' or ')' or by the omission of a required 'BEGIN' or '('. As with the extra left parenthesis, check on the syntax of control statements and declarations. The compiler tries to recover from this error by introducing a left parenthesis to match the right parenthesis in question. If the right parenthesis was merely misplaced, this will produce the message EXTRA LEFT PARENTHESIS at the end of compilation.

IMPROPER ARGUMENT OF MOD FUNCTION

The arguments of the intrinsic function MOD must be of integer type.

IMPROPER ARGUMENT OF PROCEDURE

Too many or too few arguments have been given in a FUNCTION or a procedure-call statement.

IMPROPER ARRAY DECLARATION

An error in syntax has occurred in an ARRAY declaration. Check in particular to see that all of the dimensions are specified as integer constants.

IMPROPER ASSIGNMENT OPERATION

The symbol = has appeared in relational context. The compiler will generate instructions as though the reserved word EQL had been written. The error is detected only if the misplaced character forms part of an IF or UNTIL clause.

IMPROPER ASSIGNMENT STATEMENT

An expression or a constant has occurred to the left of an '='.

IMPROPER BOOLEAN OPERAND

An attempt has been made to use a floating-point quantity as a Boolean operand. The compiler cannot detect integer quantities used as Boolean operands.

IMPROPER CARDATRON INSTRUCTION

An instruction following the CARDATRON input-output pseudo-operation does not have an operation code of 60 through 65; or has improper 'standard' unit designation.

IMPROPER CHARACTER PAIR

Two successive characters (omitting spaces) have occurred which are meaningless, for example, '+)', '=*', or ',;'. The compiler will ignore the second of these characters and continue scanning.

IMPROPER CHECK SUM

A failure has occurred in the magnetic-tape system. The compiler will reread the bad tape block repeatedly in an attempt to bring in the information correctly.

IMPROPER EMPTY SUBSCRIPT POSITION

One of the character pairs '(,' ',,' ',)' or '( )' has occurred in the wrong context.

IMPROPER EQUIVALENCE

In an external program the 82-field of an instruction having a sign of five or six was not specified on an equivalence card.

IMPROPER EQUIVALENCE CARD

An undefined name, or an equivalence number over 99, has been specified on an equivalence card.

IMPROPER FUNCTION ARGUMENT

An expression has been written as an output or program-reference argument; a label, function name, or procedure name has been used as an input or output argument; or an array has been used as a program-reference argument in a procedure call.

IMPROPER INPUT DECLARATION

An expression or constant has been used as a quantity to be read in.

IMPROPER PSEUDO-OP

An instruction with a sign of four has an incorrect operation field ($sL = 62$) code.

IMPROPER RELATION OPERATION

One of the arithmetic relational operators has occurred in the wrong context.

IMPROPER SCALE FACTOR

An attempt has been made to use a scale factor which is not an integer constant.

IMPROPER LABEL SYMBOL

A symbol other than a letter or digit has occurred in a label.

IMPROPER SUBSCRIPT

Either too many or too few subscripts have been associated with an array.

IMPROPER VARIABLE SYMBOL

An identifier has been used as a variable, in conflict with previous context which implied that it was not a variable.

MEMORY CAPACITY EXCEEDED

The object program has exceeded the memory capacity of the computer. If all array dimensions are correct, segmentation will have to be used.

MISPLACED ARITHMETIC OPERATION

One of the symbols + − · / or * has been used in the wrong context.

MISPLACED COMMA

The symbol , has been used in the wrong context.

MISPLACED DECIMAL POINT

The symbol . has been used in the wrong context.

MISSING FIELD ON SYMBOLIC CARD

A name card or equivalence card of an external program was incorrect.

MISSING FINISH CARD

The FINISH card required after the last machine-language deck was not present.

MISSING FINISH PSEUDO-OP

The machine-language deck of an external program was not terminated by a pseudo-operation for FINISH.

MISSING OPERAND

A misplaced operand may result in the error messages MISSING OPERAND and EXTRA OPERAND appearing in the printout of the compilation. The compiler expected an operand and could not find it. Check the formation of identifiers and the syntax of control statements.

MISSING NAME CARD

The first card of either an external procedure or an external statement was not a name card.

PREFIX PROCEDURE NOT DECLARED

The procedure used as a prefix on an equivalence card was not declared within the symbolic program.

UNDEFINED EXTERNAL PROCEDURE £

If an external procedure £ was declared in the symbolic program but was not defined by a machine-language deck, the first ten characters of its identifier are printed along with this error message.

UNDEFINED LABEL –£

The label £ was not defined within the symbolic program. Only the first ten characters of £ will be printed if it was an identifier. An integer is printed in its entirety.

DIAGNOSTIC AIDS TO OBJECT PROGRAM EXECUTION

It is often desirable—and sometimes necessary— to be able to inspect the intermediate results produced by a program. For instance, an iteration scheme may fail to converge in an expected time. This may be due to the slowness of the convergence, or to special numerical phenomena which often occur in iterative computation with finite numbers subject to rounding errors. By monitoring the successive values assigned to the iterated variable or array, appropriate revision may be made to the algorithm. In any extensive program a segment of the program may have been written as a loop from which—under certain circumstances—no exit is possible. By tracing the control sequence of labeled statements as they are executed, this situation may be discovered and rectified. Versatile diagnostic routines are provided in the compiler to permit access to the intermediate results of any algorithm and supervision of the control path of the program.

Another program execution aid is the provisional detection of certain error conditions which frequently occur, e.g., overflow resulting from division by zero in arithmetic operations, and unacceptable arguments to library procedures. The recognition of error conditions is performed only by the library procedures, since inclusion of the capacity to recognize spontaneously any possible error situation would necessarily reduce the efficiency of performance of a compiled program in the BURROUGHS 220.

It is also possible to obtain a listing of the machine code for the different portions of a compiled program. This listing is as useful in providing insight to the computation technique arrived at by the compiler as it is valuable in providing a diagnostic aid for detecting logical and accidental errors of a program.

These various diagnostic features of the compiler will be discussed in the following sections of this chapter.

AIDS SPECIFIED BY THE PROGRAMMER

Provision has been made in the compiler for inclusion in the object program of diagnostic routines intended to allow the programmer to monitor the action of assignment statements, trace the execution sequence of labeled statements, and obtain complete or partial dumps of the area of memory reserved for labels and identifiers.

The inclusion of these diagnostic routines is governed by the appearance of the MONITOR, TRACE, and DUMP declarators, with their associated lists, in the symbolic program. Because the labels and identifiers declared within a procedure are treated independently of those same kinds of symbols declared in the non-procedure body of a program, it is necessary to restrict the scope of these diagnostic declarations either to the procedure in which they appear or to the body of the program—exclusive of procedures—in which they are declared. This means that diagnostic declarations may appear in the body of any procedure, in the body of the program external to procedures, in both, or in neither. When diagnostic declarations are included in a program they should appear either at the beginning of the program body or immediately after the BEGIN separator of the PROCEDURE declaration in which they appear.

The inclusion of diagnostic facilities in a program considerably diminishes the rate at which the object program is executed. By suppression of some or all of the printouts specified by the diagnostic declarations—through proper employment of the PROGRAM CONTROL SWITCHES—it is possible to increase the rate of program execution. If a program is to be employed more than once, it is suggested that this program be recompiled with the diagnostic declarations and unnecessary programed halts removed.

The MONITOR Declaration

The MONITOR declaration enables the programmer to witness production of the results produced by assignment statements.

GENERAL FORM:

MONITOR 𝔐£

The *monitor list* 𝔐£ may be composed of elements which are either or both of two kinds of symbols. They may be the identifiers of simple variables, arrays, functions, or procedures, or they may be the labels of statements, subroutines, or segments. Each time an assignment is made to an identifier included in the monitor list, or any assignment is made within the scope of a label appearing in the monitor list, the first five characters of the identifier to which assignment is made are printed along with the value which has been assigned. No parentheses or other special characters may appear in the monitor list. The depression of PROGRAM CONTROL SWITCH 0 at object time will cause suppression of the monitoring action.

The DUMP Declaration

The DUMP declaration serves to declare those items which are to be exhibited when a symbolic memory dump is called for by a TRACE declaration.

GENERAL FORM:

DUMP 𝔇𝔏

The *dump list* 𝔇𝔏 may consist of elements which are either identifiers of simple variables or arrays, or which are labels of statements or subroutines. If a DUMP declaration appears as part of a procedure, then the identifiers which appear in its dump list must be only call-by-value simple variables contained in the procedure parameter list.

If a dump list of a DUMP declaration is empty, this indicates that all identifiers of simple variables or arrays, and all labels of statements or subroutines within the scope of that declaration, are to be exhibited according to the output format of the dump routine. This output format is composed of a listing under four headings. The first heading is a line which reads

LAST LABEL PASSED WAS 𝔏

where 𝔏 represents the first ten characters of the actual label which was last passed. The second heading is a line which reads

LABEL IN PROGRAM NUMBER OF TIMES EXECUTED

and beneath these separate titles are listed respectively the first ten characters of each label in alphabetical order, and a four-digit integer giving the actual number of times that label has been encountered. The third heading is a line which reads

VARIABLE IN PROGRAM VALUE

and under these respective titles are listed the first five characters of each simple variable identifier and the current value associated with that identifier. The fourth heading is a line which appears as

ARRAY 𝔤

where 𝔤 is an array identifier. Beneath this heading will appear the current values of the elements of the array designated by 𝔤. The order in which these elements appear is identical to that which would be employed to fill the array designated by 𝔤. (See *Filling an Array*, CHAPTER V.)

When a TRACE declaration has called for a symbolic dump, and a DUMP declaration was given as part of a PROCEDURE declaration, then the output format for the symbolic dump of that procedure is preceded by a heading which reads

PROCEDURE 𝔤

where 𝔤 represents the first five characters of the procedure identifier.

The TRACE Declaration

The TRACE declaration is used to cause symbolic dumps at specified points in the program.

GENERAL FORM:

TRACE 𝔗𝔏

The *trace list* 𝔗𝔏 is composed of elements which may be of two distinct forms:

First form: 𝔏

Second form: 𝔏(*n*)

where 𝔏 is a statement label and *n* is an integer constant with a maximum of four significant digits. These forms indicate respectively that every time statement label 𝔏 is encountered, or only the *n*th time statement label 𝔏 is encountered, a symbolic dump of all items declared in *all* DUMP declarations is to take place according to the output format of the dump routine. This dump will occur as the statement label 𝔏 is encountered, which is always *before* the execution of the statement labeled by 𝔏. By depressing PROGRAM CONTROL SWITCH 9, all action caused by TRACE declarations will be suppressed.

Whenever a DUMP declaration appears in a symbolic program, special provision is made to allow a comprehensive symbolic dump at any point in the program where a programed halt is planned. By depressing the RESET-TRANSFER switch following a programed halt, a comprehensive symbolic dump may be obtained of all labels and identifiers within the scope of all DUMP declarations. This dump is provided according to the output format of the dump routine. Control may be resumed in sequence by depressing the START switch, following the machine stop which occurs after the printout.

Minimum Monitoring

Minimum monitoring is another aspect of the diagnostic facilities. The library procedures have certain error messages which are dependent upon the diagnostic routines for maintenance of particular information. For instance, such error messages require the label of the last labeled statement executed, and may also require the number of times a particular label has been encountered. Whenever any of the diagnostic declarations is included in the symbolic program, provision is made automatically to maintain this information. If no diagnostic declarations appear, then this information will not be available for printout as part of the library procedure error messages. If the programmer wishes to have only this particular information maintained (mini-

mum monitoring) the inclusion of a MONITOR declaration with an empty monitor list will cause minimum monitoring to occur within the scope of that MONITOR declaration. By depressing PROGRAM CONTROL SWITCH 8, the action of minimum monitoring may be discontinued. By depressing PROGRAM CONTROL SWITCH 7, a listing of all labels within the scope of any diagnostic declaration is provided as these labels are encountered.

ERROR MESSAGES FROM LIBRARY PROCEDURES

Provision is made in the library procedures for the printing of error messages when conditions are encountered which prohibit the proper execution of that procedure. For instance, among these conditions are an overflow upon entry to the procedure, arithmetic overflow resulting from execution of the procedure, an attempt to compute the natural logarithm or extract the square root of a negative number, and an attempt to evaluate the exponential function for an absolute argument greater than the number 112.82666.

There are four separate error messages which may appear as a result of these different conditions. The most frequent of these messages is

ARITHMETIC OVERFLOW -£ (*nnnn*)

where £ is the first ten characters of the last statement label passed, and *nnnn* is an integer which has a maximum of four digits and indicates the number of times this label has been encountered during the execution of the object program. This message will occur when an overflow condition exists upon entry to a library procedure. This overflow condition may have arisen due to division by zero, or through an attempt to determine the result R of one of the following operations:

FIXED POINT	$R = \mid x_1 \pm x_2 \mid > 10^{10}$
FLOATING POINT	$R = \mid x_1 \pm x_2 \mid > 0.99999999 \times 10^{49}$
FLOATING POINT	$R = \mid x_1 \cdot x_2 \mid > 0.99999999 \times 10^{49}$
FLOATING POINT	$R = \mid x_1 / x_2 \mid > 0.99999999 \times 10^{49}$

If no diagnostic declarations are made in the symbolic program, this information about the last label encountered and the tally of label encounters is not maintained, and these portions of the error message are blank (see *Minimum Monitoring*, above).

The other library procedure error messages depend upon the appearance in the symbolic program of diagnostic declarations in the same way as does this error message. The second possible error message reads

RESULT OUT OF RANGE ℘ -£ (*nnnn*)

where ℘ is the name of the library procedure in which the result of an arithmetic operation has resulted in an overflow. The third error message is one which reads

RESULT UNDEFINED FOR ℘ -£ (*nnnn*)

This message appears whenever a condition occurs which is not meaningful for the procedure designated by ℘ to treat. For example, this message will result if the absolute argument of the inverse cosine function (ARCCOS) is greater than one. The last possible library procedure error message which may appear is of the form

RESULT ILL-DEFINED FOR ℘ -£ (*nnnn*)

This message may be printed when the result produced by the procedure ℘ is undependable in regard to numerical accuracy, or only partially satisfactory in some other respect.

OBJECT PROGRAM LISTING

By depressing PROGRAM CONTROL SWITCH 2 before or during the compilation process, a listing of the compiled object program may be obtained as the machine program is assembled. The lines appearing in this listing will be of the following general form:

llll s bbbb pp aaaa iiiii

where *llll* indicates an absolute machine address which will be the actual machine address at object time of the word defined by *s* (the sign position), *bbbb* (the variance or control field), *pp* (the operation field), and *aaaa* (the address field). The portion of the printed line indicated by *iiiii* may be a symbol having no more than five characters. This symbol is provided to assist the programmer by indicating either the kind of quantity used or the nature of the object to be achieved by the instruction word with which this symbol is associated.

Certain portions of the printed line may be absent. For instance, if in the source program an unconditional branch (e.g., GO TO £) has been declared and the statement label £ to which reference is made has not yet been encountered, the compiler does not know the appropriate address to which control must be transferred. In this case the address field of the assembled machine-instruction word for the branch instruction will be blank. When the compiler does encounter the label to which reference was made, it will insert a line into the listing which reads

llll *aaaa*

where *llll* will be seen to be the actual machine address of the instruction which appears earlier in the listing and with an address field which was of necessity left blank.

EXAMPLE:

```
COMMENT THIS PROGRAM IS INTENDED TO ILLUSTRATE
  CERTAIN FEATURES OF THE MACHINE LANGUAGE
  PRINTOUT;
BOOLEAN ABC123;
ABC123 = 0;
0200    0  0000  46  4999  ABC12
ALPHA = BETA = 1.0;
0201    0  0000  10  4998  CONST
0202    0  0000  40  4997  BETA
0203    0  0000  40  4996  ALPHA
IDENTIFIER = ALPHA + BETA;
0204    0  0000  10  4997  BETA
0205    0  0000  22  4996  ALPHA
0206    0  0000  40  4995  IDENT
GAMMA = SQRT( IDENTIFIER );
0207    0  0000  10  4995  IDENT
0208    0  0000  44  4950  SQRT
0209    0  0000  30  4950
0210    0  0000  40  4949  GAMMA
0211    0  0000  10  4949  GAMMA
0212    0  0000  24  4949  GAMMA
0213    0  0000  40  4948  TEMP
0214    0  0000  10  4949  GAMMA
0215    0  0000  24  4949  GAMMA
0216    0  0000  22  4948  TEMP
GAMMA = (GAMMA.GAMMA + GAMMA*2)/2.0;
0217    0  0002  45  0000
0218    0  0000  25  4947  CONST
0219    0  0000  40  4949  GAMMA
0220    0  0000  10  4946  CONST
0221    0  0000  13  4999  ABC12
0222    0  0101  36  0224
0223    0  0000  10  4999  ABC12
0224    0  0000  36
IF ABC123 OR NOT ABC123;
  GO TO SOMELABELLATERINTHEPROGRAM;
0225    0  0000  30  0000  SOMEL
0224                0226
0225                0226
SOMELABELLATERINTHEPROGRAM.. STOP GAMMA;
0226    0  0000  10  4949  GAMMA
0227    0  0137  00  7310
FINISH;
0228    0  9669  00  9669
0229    0  1000  60  0000
4946    0  0000  00  0001  POOL
4998    0  5110  00  0000
4947    0  5120  00  0000

COMPILED PROGRAM ENDS AT 0229
PROGRAM VARIABLES BEGIN AT 4615
```

In addition to the listing of those machine-language words assembled by the compiler from the source-language symbolic program, a listing may also be obtained of the machine instructions of external programs and the machine instructions written by the compiler onto the program tape from the library of standard procedures. The listing of machine code for the external procedures is obtained by depressing PROGRAM CONTROL SWITCH 1 prior to or during the compilation process. If the listing of machine code for library procedures is desired, then depression of PROGRAM CONTROL SWITCH 3 will produce this listing.

ORGANIZATION OF PROGRAMS
HARMONIC-BOUNDARY VALUES
SURVEY TRAVERSE CALCULATIONS
HOUSEHOLDER REDUCTION
CROUT'S METHOD

XI . . .

programs in algol

THE OVER-ALL ORGANIZATION of programs written in BAC-220 is a matter which the rules leave largely to the preference of the individual programmer. However, there are rules of precedence which must be observed regarding the interrelations among certain declarations and statements. Below is a summary of such rules, arranged in the form of a suggested program outline. If this outline is followed, the probability of introducing additional errors into the program will be greatly diminished.

COMMENT declarations—affect neither the compilation nor the object program, and may appear anywhere, except as the final statement of a compound statement.

MONITOR declaration—appears only if one or more variables are to be monitored, or if the programmer wishes to provide for minimum monitoring without other diagnostic aids. It must be the first declaration (other than a COMMENT declaration) of a symbolic program, exclusive of procedures. When used to monitor variables located inside of a procedure, the MONITOR declaration must immediately follow the initial BEGIN of the PROCEDURE declaration.

DUMP declarations—must be placed at the beginning of the source program, or immediately after the first BEGIN when a programmer requests a symbolic dump inside a PROCEDURE declaration.

Declarations of type—declare the types of any identifiers which are not floating-point by default (see CHAPTER V); they must precede use of the identifier in a statement.

ARRAY declarations—reserve storage for arrays of data; they must precede use of any variable which is an element of an array.

INPUT and OUTPUT declarations—relate the numbers to be read into the computer as input to their respective symbolic variables.

FORMAT declarations—describe the appearance of the output line or card.

PROCEDURE and FUNCTION declarations (symbolic and external)—make specific subprograms available to the compiler; they must appear prior to the first use of the subprograms.

SUBROUTINE declarations—make subprograms available to that portion of the program (procedure or main body) in which they are declared. It is not necessary that the declaration be declared before it is called.

Statements—assignment statements, procedure-call statements, and control statements constituting the main program, a PROCEDURE declaration, or a subroutine.

FINISH declaration—must appear at the end of the symbolic program.

Machine-language programs—may be used in conjunction with a symbolic program. The latter must contain declarations of all the machine-language programs so used. These programs are included after the FINISH declaration of the symbolic program, and must themselves be terminated by a second FINISH declaration.

EXAMPLES OF PROGRAMS

The following complete programs are presented as illustrations of the rules delineated in this manual.

J. G. Herriot, of Stanford University, has written the following program to determine an approximation of harmonic-boundary values, using orthonormal functions.

```
COMMENT THIS PROGRAM FIRST CONSTRUCTS A SET OF
  ORTHONORMAL FUNCTIONS AND THEN USES THEM TO
  FIND AN APPROXIMATION TO THE SOLUTION OF A
  HARMONIC BOUNDARY-VALUE PROBLEM;
COMMENT WE FIRST CONSTRUCT THE ORTHONORMAL
  FUNCTIONS;
INTEGER I, J, K, L, M, N, NU, TH;
ARRAY R(29), HFN(29), DSUM(24), HFCN(5), HFCEN(6),
  FA(25,25), A(25,25), B(25,25), HA(47), HAA(24);
INPUT DATA (FOR I = (1,1,29); R(I)), DIMEN(N);
OUTPUT FRESULTS (FOR I = (1,1,N); FOR J = (1,1,N); FA(I,J)),
  ARESULTS (FOR I = (1,1,N); FOR J = (1,1,N); A(I,J)),
  BRESULTS (FOR I = (1,1,N); FOR J = (1,1,N); B(I,J)),
  COEFFS (FOR NU = (4,4,N - 1); HA (2NU - 1)),
  HFNRES (FOR K = (1,1,29); HFN(K)),
CRES(CONST), HFCNRES (TH, FOR K = (1,1,5); HFNC(K)),
  HFCENRES (TH, FOR K = (1,1,6); HFCEN(K));
FORMAT VECTOR (B8,6F16.8,W0),
  FTITLE (B48,*FRESULTS,FA(I,J)*,W3,W2),
  ATITLE (B48,*ARESULTS,A(I,J)*,W3,W2),
  COEFTITLE (B30,*HA(8NU - 1)*,W2),
  BDYVALUES (B42,*PRELIMINARY BOUNDARY VALUES*,W3,
    W2),
  CBDYVALUES (B43,*CORRECTED BOUNDARY VALUES*,W2),
  CONTITLE (B50,*CONSTANT*,W2),
  TABLE (B8,I2,B6,6F16.8,W0),
  TABLEHEAD (S40, *THE VALUES OF H(RHO,TH) IN B*, W3,
    W2),
  TABLELINE (B13,*RHO*,B6,*0.5*,B13,*1.0*,B13,*1.5*,B13,
  *2.0*, B13,*2.5*,B13,*3.0*,W0),
  TABLETH (B8,*TH*,W0);
START..READ (;;DATA); RDIM..READ (;;DIMEN);
  FOR I = (1,1,N); FOR J = (I,4,N);
BEGIN L = I - J; K = I + J;
  SUM = R (1)*K + 1.5.R(18)*K.COS(0.59341195.L)
    + 0.5.R(29)*K.COS(0.78539816.L);
  FOR M = (2,1,17);
  SUM = SUM + 2.0.R(M)*K.COS((M - 1).0.034906585.L);
  FOR M = (19,1,28);
  SUM = SUM + R(M)*K.COS((0.59341195 + (M - 18)
    .0.017453293).L);
  FA(I,J) = (8.0/K).0.017453293.SUM END;
    WRITE (;;FTITLE);
WRITE (;; FRESULTS, VECTOR);
  FOR J = (1,1,N); B(1,J) = FA(1,J);
  FOR I = (2,1,N);
    BEGIN FOR J = (1,1,I - 1);
      B(I,J) = -B(J,I)/B(J,J);
    FOR J = (I,1,N);
      BEGIN B(I,J) = FA(I,J);
    FOR K = (1,1,I - 1);
      B(I,J) = B(I,J) + B(I,K).B(K,J) END;
  FOR J = (1,1,I - 1);
      B(I,J) = B(I,J).SQRT(B(J,J)/B(I,I)) END;
FOR I = (1,1,N); B(I,I) = 1.0/(SQRT(B(I,I)).I);
WRITE (;;BTITLE);
WRITE (;;BRESULTS, VECTOR);
FOR I = (1,1,N); FOR J = (1,1,N); A(I,J) = 0;
  A(1,1) = B(1,1);
FOR I = (2,1,N);
  BEGIN FOR J = (1,1,I - 1);
    BEGIN A(I,J) = 0;
  FOR K = (J,1,I - 1);
  A(I,J) = A(I,J) + B(I,K).A(K,J) END;
    A(I,I) = B(I,I) END; WRITE (;;ATITLE);
WRITE (;;ARESULTS, VECTOR);
COMMENT NOW CONSTRUCT THE APPROXIMATION TO
THE SOLUTION;
  FOR J = (4,4,N - 1);
    BEGIN DSUM(J) = 0;
  FOR M = (1,1,17);
    DSUM(J) = DSUM(J) + (R(M)*2 + R(M + 1)*2).
    (R(M + 1)*J.SIN(M.0.034906585.J)
    -R(M)*J.SIN((M - 1).0.034906585.J));
  FOR M = (18,1,28);
  DSUM(J) = DSUM(J) + (R(M)*2 + R(M + 1)*2.(R(M + 1)
    *J.SIN((0.59341195 + (M - 17).0.017453293).J)
    - R(M)*J.SIN((0.59341195 + (M - 18).0.017453293)
      .J))) END;
  FOR NU = (4,4,N - 1); BEGIN HA(2NU - 1) = 0;
    FOR J = (4,4,NU);
    HA(2NU - 1) = HA(2NU - 1) + A(NU,J).DSUM(J);
    HA(2NU - 1) = 4.0.HA(2NU - 1) END;
      WRITE (;;COEFTITLE);
  WRITE (;;COEFFS, VECTOR);
  FOR J = (4,4,N - 1); BEGIN HAA(J) = 0;
    FOR NU = (J,4,N - 1);
  HAA(J) = HAA(J) + HA(2NU - 1).A(NU,J) END;
FOR M = (1,1,18); BEGIN HFN(M) = 0;
  FOR J = (4,4,N - 1);
  HFN(M) = HFN(M) + HAA(J).R(M)*J.COS((M - 1)
    .0.034906585.J) END;
FOR M = (19,1,29); BEGIN HFN(M) = 0;
FOR J = (4,4,N - 1);
  HFN(M) = HFN(M) + HAA(J).R(M)*J.COS((0.59341195
    + (M - 18).0.017453293).J) END;
WRITE (;;BDYVALUES);
WRITE (;;HFNRES, VECTOR);
AVT = 0;
FOR M = (1,1,29); AVT = AVT + R(M)*2 - HFN(M);
  CONST = AVT/29.0; WRITE (;;CONTITLE);
  WRITE (;;CRES, VECTOR);
FOR M = (1,1,29); HFN(M) = CONST + HFN(M);
WRITE (;;CBDYVALUES);
```

```
WRITE (;;HFNRES, VECTOR);
FOR I = (1,1,5); BEGIN TH = 5.(I – 1);
  FOR J = (1,1,5);
BEGIN HFCN(J) = CONST;
FOR M = (4,4,N – 1);
HFCN(J) = HFCN(J) + HAA(M).(0.5.J)*M.
COS((I – 1).0.087266463.M) END;
  WRITE(;;TABLEHEAD);
  WRITE(;;TABLELINE);
  WRITE(;;TABLETH);
WRITE (;;HFCNRES, TABLE) END;
FOR I = (6,1,10); BEGIN TH = 5.(I – 1);
  FOR J = (1,1,6);
BEGIN HFCEN(J) = CONST;
FOR M = (4,4,N – 1);
HFCEN(J) = HFCEN(J) + HAA(M).(0.5.J)*M.COS((I – 1)
    .0.087266463.M) END;
WRITE (;;HFCENRES, TABLE) END;
STOP 1234;
GO TO RDIM;
  FINISH;
```

The program which follows is one for survey traverse calculations.

```
COMMENT SURVEY TRAVERSE CALCULATIONS;
TRACE ANGLE;
DUMP EW, NSC, CD;
INTEGER I, J, K, SURVEY, D(), M() , S() , Q() , N;
FUNCTION LENGTH(X,Y) = SQRT(X*2 + Y*2);
ARRAY D(200), M(200), S(200), Q(200), MD(200), NS(200),
  EW(200), CNS(201), CEW(201);
  START.. READ (;;IDENT); TMD = 0; TNS = 0; TEW = 0;
FOR I = (1,1,N); BEGIN
READ (;;STATION); IF I NEQ K; STOP K;
Z = (60(60D(I) + M(I)) + S(I))/6.48**5;
SWITCH Q(I), (QUAD1, QUAD2, QUAD3, QUAD4);
QUAD1.. Z = 0.5 – Z; GO TO ANGLE;
QUAD2.. Z = 1.5 + Z; GO TO ANGLE;
QUAD3.. Z = 0.5 + Z; GO TO ANGLE;
QUAD4.. Z = 1.5 – Z;
ANGLE.. ALPHA = 3.1415927Z;
NS(I) = MD(I)SIN(ALPHA); TNS = TNS + NS(I);
EW(I) = MD(I)COS(ALPHA); TEW = TEW + EW(I);
TMD = TMD + MD(I) END;
ERROR = LENGTH (TNS, TEW); WRITE (;;TITLE, F1);
NSC = –TNS/TMD; EWCF = –TEW/TMD; TCD = 0;
  TCNS = 0; TCEW = 0;
FOR I = (1,1,N); BEGIN
CNS(I) = NS(I) + MD(I).NSCF; TCNS = TCNS + CNS(I);
CEW(I) = EW(I) + MD(I).EWCF; TCEW = TCEW + CEW(I);
CD = LENGTH(CNS(I), CEW(I)); TCD = TCD + CD;
WRITE (;;ANSWERS,F2) END;
CNS(N + 1) = CNS(1); CEW(N + 1) = CEW(1); SUM = 0;
FOR I = (1,1,N); SUM = SUM + (CNS(I + 1) – CNS(I))
  (CEW(I + 1) + CEW(I));
SQFT = ABS(SUM)/2; ACRES = SQFT/43560;
WRITE (;;TOTALS, F3); GO TO START;
INPUT IDENT(SURVEY, N), STATION(K,D(I), M(K), S(I),
  Q(I), MD(I));
OUTPUT TITLE(SURVEY, N, ERROR),
ANSWERS (I, D(I), M(I), S(I), Q(I), MD(I), CD, CNS(I), CEW(I)),
TOTALS (TMD, TCD, TCNS, TCEW, SQFT, ACRES);
FORMAT F1(*SURVEY*, I8, B5, *NUMBER OF LEGS*, I5,
  *CLOSURE ERROR*, X9.2, W1, *LEG*, B
5, *ANGLE*, B7, *MEASURED*, B5, *CORRECTED*, B3,
  *NORTH-SOUTH EAST-WEST*, W6, *NO. DD
MM SS Q DISTANCE DISTANCE DISPLACEMENT
  DISPLACEMENT*, 2W), F2,(I3, I5, 2I3, I2, 4X13.2, W),
F3(B6, *TOTALS*, B4, 4X13.2,W4, *AREA OF TRAVERSE*,
  X13.2,*SQUARE FEET*,
X13.2,*ACRES*, W6);
FINISH;
```

The short program which follows is for a reduction of a square matrix to tridiagonal form, using the method of Householder.

```
COMMENT HOUSEHOLDER REDUCTION TO TRIDIAGONAL
  FORM;
INTEGER I, J, K, L, R, N; ARRAY A (50,50), X(50), P(50);
INPUT ELEMENT (I,J,Q); OUTPUT AOUT (A(R,R)),
  BOUT (–0.5/S);
FORMAT AF(B10, X10.5, W), BF(B40, X10.5,W);
N = 5;
IN.. READ(;;ELEMENT); IF I NEQ 0; BEGIN A(I,J) = Q;
  GO TO IN END;
FOR R = (1,1,N – 1); BEGIN WRITE (;;AOUT, AF); L = R + 1;
S = 0; FOR J = (L,1,N); S = S + A(R,J)*2;
  S = SIGN (A(R,L))/2SQRT(S);
WRITE (;;BOUT, BF);
X(L) = SQRT(0.5 + A(R,L).S); S = S/X(L);
FOR J = (R + 2,1,N); X(J) = S.A(R,J);
FOR J = (R,1,N); BEGIN S = 0; FOR K = (L,1,N);
S = S + A(MIN(J,K), MAX(J,K)).X(K); P(J) = S END;
S = 0; FOR J = (L,1,N); S = S + K(J).P(J);
FOR J = (L,1,N); P(J) = P(J) – S.X(J);
FOR J = (L,1,N); FOR K = (J,1,N); A(J,K) = A(J,K) – 2(X(J).
  P(K) + X(K).P(J)) END;
WRITE (;;AOUT, AF); STOP; GO TO IN;
FINISH;
```

The program below has been written by G. Forsythe, of Stanford University. It solves a set of linear equa-

tions of the form **Ay = B**, using Crout's method with interchanges.

```
COMMENT FORSYTHE PROGRAM;
PROCEDURE PRODUCT (;N, A( ), P, E);
BEGIN
COMMENT THIS FORMS THE PRODUCT OF ARBITRARY FLOAT-
    ING NUMBERS A(I),
  FOR I = (1,1,N). EXPONENT OVERFLOW OR UNDERFLOW IS
  PREVENTED. THE ANSWER IS P TIMES 10*E, WHERE E IS 0
  IF POSSIBLE. IF E NEQ 0, THEN WE NORMALIZE P SO THAT
  0.1 LEQ ABS(P) LSS 1.0;
INTEGER E, F, I, K, N;
Q = 1.0**-10; F = 10;
FOR I = (1,1,N);
BEGIN IF A(I) EQL 0.0;
  BEGIN P = 0.0; E = 0; RETURN END;
  IF ABS(A(I) ) LEQ 1.0;
  BEGIN F = F - 20; Q = Q.(10.0*20) END;
  Q = Q.A(I); X = ABS(Q);
  FOR K = (-10,1,10), (-11, -1, -41), (11,1,41);
  IF ( (10.0*K LEQ X) AND (X LSS 10.0*(K + 1) ) );
BEGIN Q = Q.(10.0*(-10 - K) ); F = F + K + 10; GO TO 1
    END;
1.. END;
  IF ( ( (-40) LEQ F) AND (F LEQ 58) );
  BEGIN P = (Q.(10.0*9) ).(10.0*(F - 9) ); E = 0; RETURN
    END;
  P = Q.(10.0*9); E = F - 9; RETURN END PRODUCT ( );
PROCEDURE INNERPRODUCT (S,F,U( ), V( ) );
BEGIN COMMENT THIS FORMS THE INNER PRODUCT OF THE
    VECTORS U(I) AND V(I) FOR I = (S,1,F);
  INTEGER I, S, F; SUM = 0.0;
  FOR I = (S,1,F); SUM = SUM + U(I).V(I);
  INNERPRODUCT( ) = SUM;
  RETURN END INNERPRODUCT( );
PROCEDURE CROUT4(;N, A (,), B( ), Y( ), PIVOT( ), DET, EX7;
SINGULAR, IP( ) );
BEGIN COMMENT THIS IS CROUTS METHOD WITH INTER-
CHANGES, TO SOLVE AY = B AND OBTAIN THE TRIANGULAR
DECOMPOSITION. IP( ) STANDS FOR AN INNER PRODUCT
ROUTINE THAT MUST BE AVAILABLE WHEN CROUT4( ) IS
CALLED. ALSO, PRODUCT ( ) MUST BE AVAILABLE. THE
DETERMINANT OF A IS COMPUTED IN THE FORM DET TIMES
10*EX7, WHERE EX7 IS 0 IF POSSIBLE. IF EX7 NEQ 0, THEN
WE NORMALIZE DET WITH 0.1 LEQ ABS(DET) LSS 1;
INTEGER K, I, J, IMAX, N, PIVOT; INTEGER EX7; INT = 1.0;
FOR K = (1,1,N):
BEGIN TEMP = 0; FOR I = (K,1,N);
  BEGIN A(I,K) = A(I,K) - IP(1, K - 1, A(I, ), A( ,K) );
  IF ABS(A(I,K) ) GTR TEMP;
  BEGIN TEMP = ABS(A(I,K) ); IMAX = I END END;
  PIVOT(K) = IMAX;
COMMENT WE HAVE FOUND THAT A(IMAX,K) IS THE LARGEST
PIVOT IN COL K. NOW WE INTERCHANGE ROWS K AND IMAX;
IF IMAX NEQ K; BEGIN INT = -INT; FOR J = (1,1,N);
  BEGIN TEMP = A(K,J); A(K,J) = A(IMAX,J);
  A(IMAX,J) = TEMP END;
  TEMP = B(K); B(K) = B(IMAX); B(IMAX) = TEMP END;
COMMENT NOW FOR THE ELIMINATION;
IF A(K,K) EQL 0.0;
BEGIN DET = 0.0; EX7 = 0; GO TO SINGULAR END;
  FOR I = (K + 1,1,N);
  BEGIN XX = A(I,K); XY = A(K,K); X = 1.0; X = X.X;
    A(I,K) = XX/XY END;
  FOR J = (K + 1,1,N); A(K,J) = A(K,J) - IP(1, K - 1, A(K,),
    A(,J) );
  B(K) = B(K) - IP(1, K - 1, A(K,), B( ) ) END;
  FOR I = (1,1,N); Y(I) = A(I,I);
  PRODUCT (; N, Y( ), DET, EX7); DET = INT. DET;
COMMENT NOW FOR THE BACK SUBSTITUTION;
FOR K = (N,-1,1);
BEGIN XX = B(K) - IP(K + 1, N, A(K,), Y( ) ); XY = A(K,K);
  X = 1.0; X = X.X; Y(K) = XX/XY END; RETURN END CROUT4
    ( );
PROCEDURE SOLV2(; N, B(,), C( ), PIVOT( ), Z( ); IP( ) );
BEGIN
COMMENT IT IS ASSUMED THAT A MATRIX A HAS ALREADY
BEEN TRANSFORMED INTO B BY CROUT, BUT THAT A NEW
COLUMN C HAS NOT BEEN PROCESSED. SOLV2( ) SOLVES
THE SYSTEM BZ = C. AN INNERPRODUCT PROCEDURE MUST
BE USED WITH SOLV2( );
    INTEGER K, N, PIVOT;
  FOR K = (1,1,N);
  BEGIN TEMP = C(PIVOT(K) ); C(PIVOT(K) ) = C(K);
    C(K) = TEMP; C(K) = C(K) - IP(1, K - 1, B(K,), C( ) )
    END;
  FOR K = (N, -1, 1); Z(K) = (C(K) - IP(K + 1, N, B(K),
    Z( ) ) )/B(K,K);
  RETURN END SOLV2( );
COMMENT FORSYTHE TEST CROUT US169 EXT 2274;
FORMAT FRMTFL(W0, (6F19.8, W0) );
FORMAT FRMTFX(W0, (6I19, W0) );
INTEGER PIVOT ( ); INTEGER EX; INTEGER I, J, N;
ARRAY A(70, 70, B(70), Y(70), C(70), PIVOT(70);
INPUT DATA(N, FOR I = (1,1,N); (FOR J = (1,1,N); A(I,J),
  B(I) ) );
INPUT VECTOR(N, FOR I = (1,1,N); C(I) );
START.. READ(; ; DATA); READ(; ; VECTOR); OUTPUT ORDER
  (N);
OUTPUT DATAO (FOR I = (1,1,N); (FOR J = (1,1,N); A(I,J),
  B(I) ) );
OUTPUT VECTORO (FOR I = (1,1,N); C(I) );
WRITE (; ; ORDER, FRMTFX);
WRITE (; ; DATAO, FRMTFL);
WRITE (; ; ORDER, FRMTFX);
WRITE (; ; VECTORO, FRMTFL);
CROUT4 (; N, A(,), B( ), Y( ), PIVOT( ), DET, EX:
```

```
  SINGULAR, INNERPRODUCT( ) );
WRITE (; ; DATAO, FRMTFL);
OUTPUT ANSWER (FOR I = (1,1,N); Y(I) );
OUTPUT PIVOTO (N, FOR I = (1,1,N); PIVOT (I) );
OUTPUT DETO (DET);
OUTPUT EXPO (EX);
WRITE (; ; PIVOTO, FRMTFX);
WRITE (; ; ANSWER, FRMTFL);
WRITE (; ; DETO, FRMTFL);
WRITE (; ; EXPO, FRMTFX);
SOLV2 (; N, A(,), C( ), PIVOT( ), Y( ); INNERPRODUCT( ) );
WRITE (; ; VECTORO, FRMTFL);
WRITE (; ; ANSWER, FRMTFL);
GO TO START; SINGULAR..WRITE (; ; FRMTSI);
FORMAT FRMTSI (W0, *SINGULAR*, W0); GO TO START;
    FINISH;
```

APPENDIX A

operating instructions for the BAC-220 generator program

READYING THE EQUIPMENT

TWO REELS OF MAGNETIC TAPE are required to generate this compiler: the BAC-220 Generator Tape and a tape on which the final version of the compiler is to be stored. The procedure to be followed is:

First: Mount the generator on TAPE STORAGE UNIT 10, and place the unit in the NOT-WRITE status.

Second: On TAPE STORAGE UNIT 2, mount the tape on which the compiler is to be written, and set the unit to the WRITE status. Either an edited or a 100-word preblocked tape may be used. PROGRAM CONTROL SWITCH 1 must be put in the ON position whenever the compiler is to be written on an edited tape.

Third: Load the CARD READER with the generator input deck, which is constructed in accordance with the specifications given within this appendix.

RUNNING THE GENERATOR PROGRAM

To execute the generator program, the following steps must be performed:

First: Ensure that the equipment has been readied; and

Second: If the BAC-220 GENERATOR CALLOUT routine is maintained on cards, execute a CRD instruction (0 *u*000 60 0000).

The generator program will now run automatically until the end of generation, or until interrupted by an error message from the SUPERVISORY PRINTER. If the latter condition occurs, the computer will halt with a display of 7310 00 1370 in the C register, and the generating process must be repeated.

COMPOSITION OF GENERATOR INPUT DECK

The generator input deck may include a variety of symbolic statements that completely describe the characteristics of the compiler desired. The specific generator input statements required for any version of the compiler are outlined within this appendix. These statements are categorized into six groups for ease of understanding. Each card contains one statement punched in a free format; the only exception is column 1, which must contain the digit two. Digits required for the quantities to be used in the blanks below do not require leading zeros.

Compiler Version Statements

```
GENERATE STANDARD VERSION OF BAC-220;
GENERATE HSP VERSION OF BAC-220;
GENERATE PAPER TAPE VERSION OF BAC-220;
```

Corrections Statement

```
CORRECTIONS·
```

System Environment Statements

```
COPY COMPILER FOR UNIT__ LANE__ ;
MACHINE-LANGUAGE OUTPUT ON UNIT__ LANE__ ;
CARD READER IS UNIT__ ;
CARD PUNCH IS UNIT__ ;
PRINTER IS UNIT__ ;
MEMORY SIZE IS__ ;
```

Input-Output Facilities Statements

DELETE INPUT FORMAT 0;
DELETE INPUT FORMAT 1;
DELETE OUTPUT FORMAT 0;
DELETE OUTPUT FORMAT 1;
INPUTMEDIA;
OUTPUTMEDIA;

Miscellaneous Option Statements

POSITION GENERATED TAPE FORWARD__ BLOCKS;
SET SCANNING FOR COLUMNS__ THROUGH__ ;
PUNCH LIBRARY__ INSTRUCTIONS PER CARD;
COMMENT__ ;
SUPPRESS OK HALT;
COMPILATION BEGINS AT__ ;
VARIABLES END AT__ ;
PROCESS LIBRARY;

FINISH Statement

FINISH;

STANDARD VERSION OF BAC-220

The standard version of the compiler (see combination 1, TABLE 1) utilizes the BURROUGHS library of standard procedures, and requires both CARDATRON input and output for transmittal of information

To generate a standard version of BAC-220, the deck illustrated in FIGURE 1 is required. A brief description of the various elements comprising this deck is given below.

The BAC-220 Generator Callout Deck

This deck is identical with the callout used for the standard compiler, except that the odd lane is selected rather than the even lane.

Compiler Version Statements

Statements from this group identify the type of compiler to be generated. One and only one card from this group must appear in every deck.

For generating the standard version, the following compiler version statement is used:

GENERATE STANDARD VERSION OF BAC-220;

For generating non-standard versions (discussed below), one of the following compiler version statements is to be used:

GENERATE HSP VERSION OF BAC-220;

The above statement is applicable whenever the HIGH-SPEED PRINTER is to be used for output at compile time, and compile time input is through CARDATRON.

GENERATE PAPER TAPE VERSION OF BAC-220;

The above statement is applicable whenever paper tape is to be used for input of source language at compile time. It also provides that external programs be introduced by means of paper tape.

GENERATE STANDARD VERSION OF BAC 220;

Use of the above statement is made when generating all versions which use CARDATRON for both input and output at compile time. Therefore, it is used even when some medium other than CARDATRON is desired at object time.

	COMPILE TIME		OBJECT TIME		REFER TO
COMBINATION	INPUT	OUTPUT	INPUT	OUTPUT	FIGURE NO.
1	CARDATRON	CARDATRON	CARDATRON	CARDATRON	1
2	CARDATRON	HSP	CARDATRON	HSP	3
3	Paper Tape	SUPERVISORY PRINTER	Paper Tape	SUPERVISORY PRINTER	4
4	CARDATRON	CARDATRON	CARDATRON	HSP	5
5	Paper Tape	CARDATRON	CARDATRON	HSP	6
6	Paper Tape	CARDATRON	Paper Tape	CARDATRON	7
7	CARDATRON	HSP	CARDATRON	CARDATRON	8

TABLE 1—SOME INPUT-OUTPUT COMBINATIONS

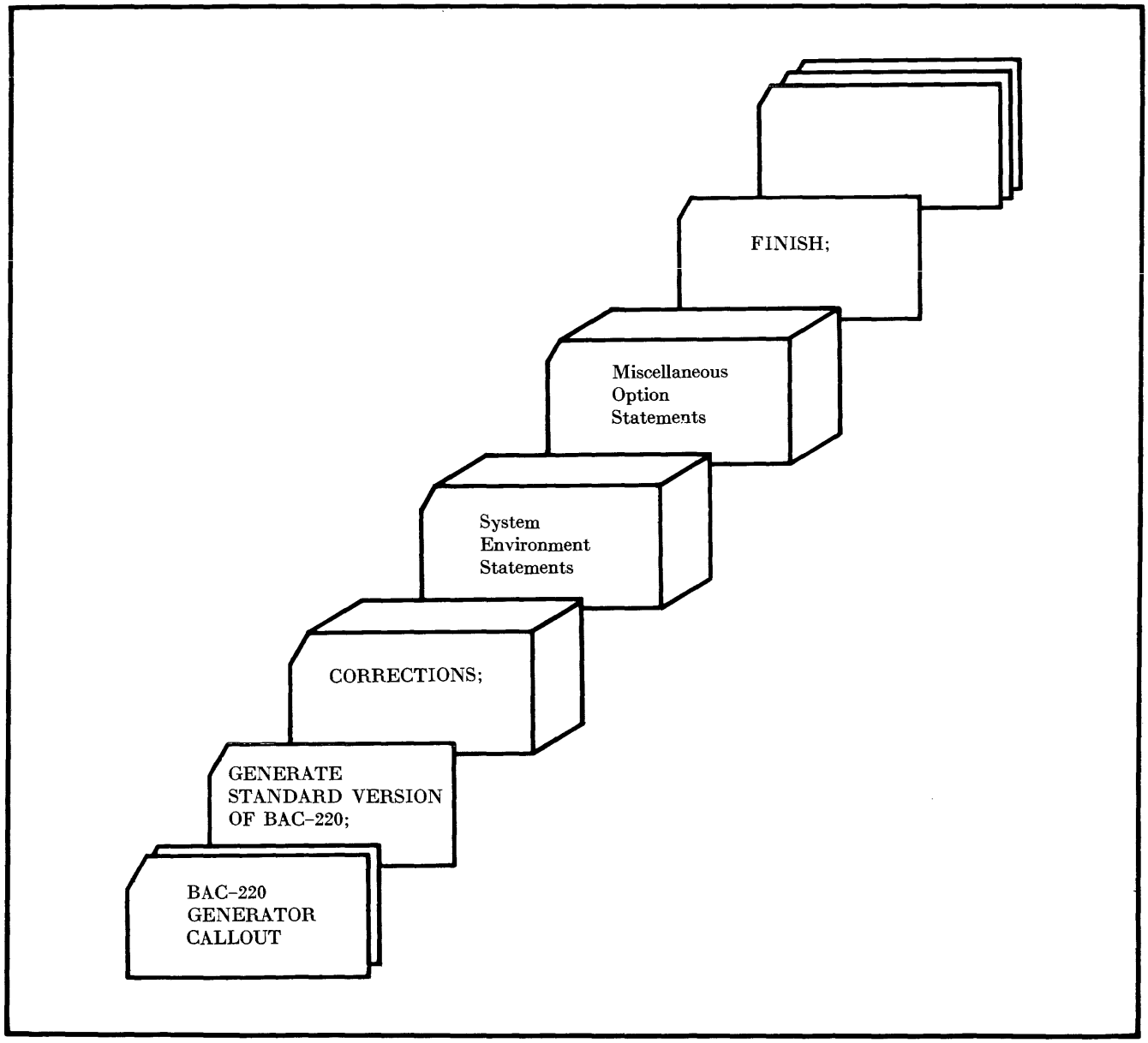


FIGURE 1. Generator Input Deck to Create Standard Version (Combination 1)

Corrections Statement

CORRECTIONS;

The corrections statement must immediately follow the compiler version statement.

The cards for this section of the deck are prepared by the user from a listing provided by the BURROUGHS CORPORATION. A complete current list can be found in *Supplemental Material for Use with the BAC-220 Generator*. Whenever this listing changes, an updated version will be sent to all users.

System Environment Statements

All versions of BAC-220 require two magnetic-tape units, one for the compiler itself and another to serve as a scratch tape for the compiler. The statement

COPY COMPILER FOR UNIT__ LANE__ ;

indicates the lane and magnetic-tape unit desired for the compiler at compile time. The compiler utilizes the working tape to store the object program. This tape must be preblocked into 100-word blocks in both lanes.

The statement

MACHINE-LANGUAGE OUTPUT ON UNIT__ LANE__ ;

is used to indicate the lane and unit designation for the scratch tape at compile time.

The statements

CARD READER IS UNIT__ ;
CARD PUNCH IS UNIT__ ;
PRINTER IS UNIT__ ;

specify the CARDATRON unit designations desired at compile time and object time. The statement

MEMORY SIZE IS__ ;

specifies the memory size of the system on which the compiler is to be used. The minimum memory size is 5,000 words. Memory size must be a multiple of one hundred.

These cards may be placed in the deck in any sequence relative to each other. The entire group of statements may be omitted; if so, a compiler will be produced with the following characteristics:

Compiler tape is produced for use on TAPE STORAGE UNIT 2, lane 00

Compiler will use lane 00 of TAPE STORAGE UNIT 1 for scratch tape

CARDATRON input will be unit 1

CARD PUNCH will be unit 2

Printer will be unit 2

Memory size will be 5,000 words.

Miscellaneous Option Statements

The statements in this group serve the following functions:

The statement

POSITION GENERATED TAPE FORWARD__ BLOCKS;

is used when the compiler program is to be placed on a system tape containing other programs. In such a case, it is unlikely that the compiler will be the first program on the tape. Consequently, the tape must be positioned forward to the appropriate place before the newly generated compiler can be written on it.

The statement

SET SCANNING FOR COLUMNS__ THROUGH__ ;

enables the format of the source-language card deck to be organized in a variety of ways. It specifies which contiguous card columns of the source deck are to be scanned.

The statement

PUNCH LIBRARY__ INSTRUCTIONS PER CARD;

produces a punched-card deck containing all procedures in the standard or non-standard BAC-220 library. A maximum of six instructions per card may be punched. A list of the standard procedures is contained in APPENDIX G.

The statement

COMMENT__ ;

is used to insert explanatory remarks in the generator input deck. This statement is formulated in accordance with the same rules which govern the COMMENT declaration in the source program (see CHAPTER V).

When compilation has been completed successfully, the compiler program normally comes to an OK halt. If immediate execution of the object program is desired instead, the statement

SUPPRESS OK HALT;

will cause the compiler to be produced without this halt.

Location 0200, which is the standard starting location of the object program, may be altered by means of the statement

COMPILATION BEGINS AT__ ;

The only restrictions are that the address at which the compiler program is specified to begin must not be less than 0200 and must be a multiple of one hundred. The memory area below the specified starting location is left intact by the memory clear routine found in the object program loader.

The effect of the statement

VARIABLES END AT__ ;

is to reserve memory space at object time between the maximum memory cell of the system and the location specified by the statement. This location must also be given as a multiple of one hundred.

The inclusion of the statement

PROCESS LIBRARY;

indicates that an extension of the standard library of procedures is desired; therefore, it is never used for generation of the standard version of BAC-220.

The miscellaneous option statements may be used in any sequence relative to the cards in this group, with one exception: if the PROCESS LIBRARY statement is used, it must be the last statement of this group. If

no miscellaneous option statements are included in the input deck, the generator program will produce a compiler conforming to the following specifications:

The compiler program will start with the *first* block of information on the compiler tape

Scanning is set for columns 2 through 72, inclusive

OK halt is not suppressed

The library of procedures is not punched out

Compilation of object program will begin at location 0200

Program variables end at the maximum cell available (e.g., 4999 in a 5,000-word system)

Compiler will be generated with the standard library of procedures.

FINISH Statement

A FINISH card must be used to terminate the input to the generator program. Three blank cards, or any number of 'reject' cards, follow the FINISH card.

NON-STANDARD VERSIONS OF BAC-220

A non-standard version of the compiler, by definition, is one which uses an extended or modified library of procedures, at least one non-CARDATRON input-output routine, or both.

To generate a non-standard version of the compiler, it is first necessary to generate a standard version and punch a standard library deck. This is accomplished by including in the input to the generator, the PUNCH LIBRARY__INSTRUCTIONS PER CARD statement from the miscellaneous option statement group, as illustrated in FIGURE 2. After the standard library

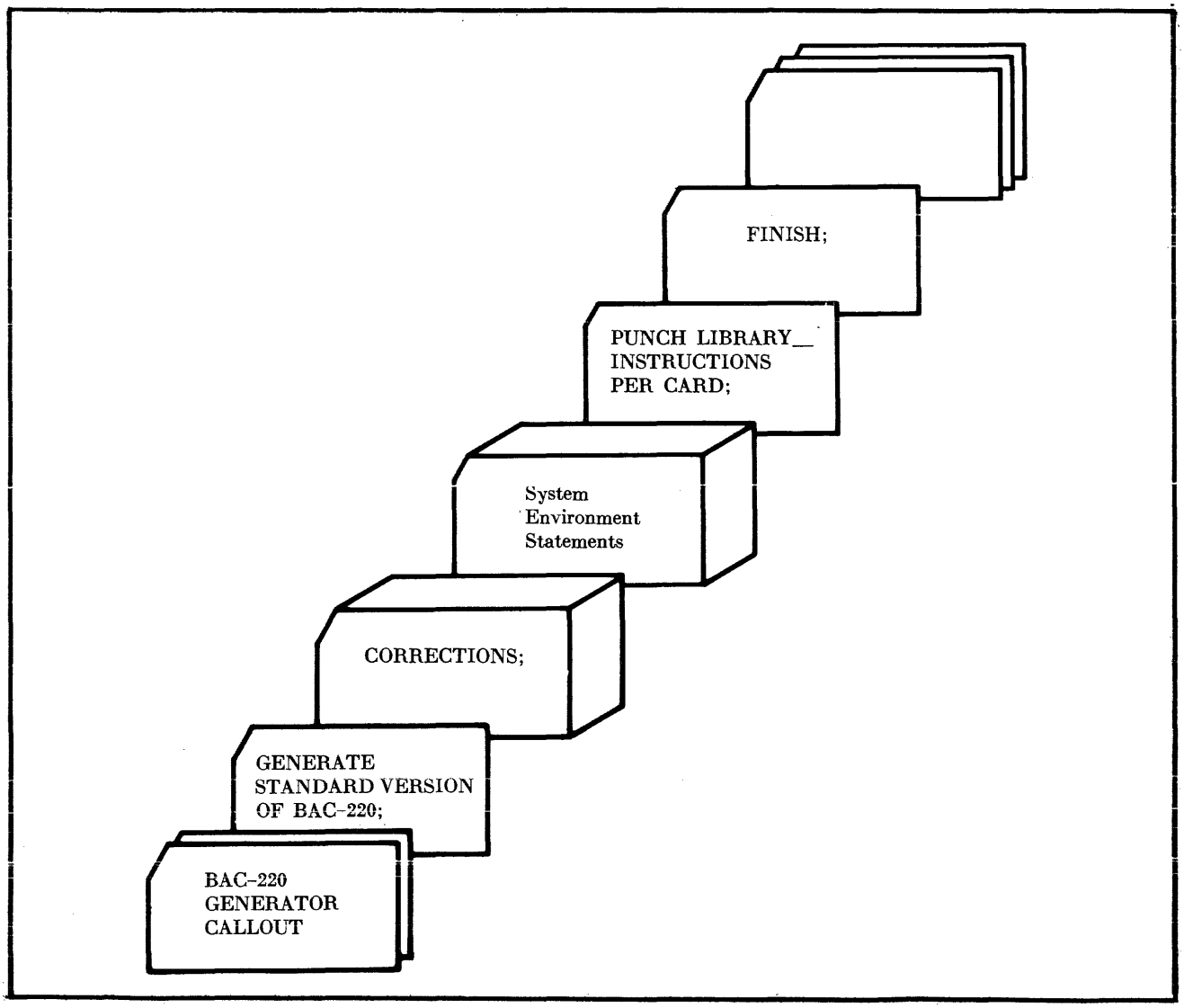


FIGURE 2. Statements Required for Standard Library Deck

has been punched, it may be modified to accommodate the particular needs of an installation. Other machine-language procedures may be added to the standard library deck for future convenience, and some of the standard routines in this deck may be replaced by non-standard input-output routines written specifically for the individual system configuration. The new non-standard library of machine-language procedures is processed by the generator and then written on the compiler tape.

The definition of the non-standard compiler requires that the programmer be familiar with the functions of the input-output facilities statements, discussed below.

Input-Output Facilities Statements

In general, the inclusion of any one statement from this group means that at least one input-output procedure is to be integrated with the compiler. The following statements

```
DELETE INPUT FORMAT 0;
DELETE INPUT FORMAT 1;
DELETE OUTPUT FORMAT 0;
DELETE OUTPUT FORMAT 1;
```

delete all CARD READ FORMAT LOAD and CARD WRITE FORMAT LOAD commands distributed throughout the compiler, with one exception: the card punch format; it is loaded regardless of the output device used. The second word of each of these statements specifies whether the non-standard routine is to affect input or output. The integer constant 0 or 1 appended to the end of one of these statements indicates whether the procedure is to be used at compile time or object time, respectively.

The routine to be used at compile time is placed in the generator input deck after either of the following statements

```
INPUTMEDIA;
OUTPUTMEDIA;
```

depending on its function. Whenever the statement

```
DELETE INPUT FORMAT 0;
```

is used, there must also be the statement

```
INPUTMEDIA;
```

followed by its associated procedure. The converse, however, is not necessarily true if one wishes to use the standard formats supplied by the compiler. Similarly, the statement

```
DELETE OUTPUT FORMAT 0;
```

may be necessary, but it is not sufficient unless used together with the statement

```
OUTPUTMEDIA;
```

The following statements

```
DELETE INPUT FORMAT 1;
DELETE OUTPUT FORMAT 1;
```

indicate that the standard REED or RITE library procedures, or both, are to be replaced by either non-CARDATRON routines, or by procedures not using the standard formats provided in the compiler. Therefore, these special procedures must be placed in the modified library deck at the end of the generator input deck.

High-Speed Printer Version of BAC-220

A compiler can be generated which utilizes an on-line HIGH-SPEED PRINTER at both compile time and object time (see combination 2, TABLE 1). The procedure to be followed is outlined below:

First: Generate a standard version of BAC-220 to produce a standard library deck. Note that the PUNCH LIBRARY—INSTRUCTIONS PER CARD statement must be included among the miscellaneous option statements.

Second: Replace the RITE procedure in the standard library deck with a special RITE written by the user to specifically produce output of results, symbolic dumps, and diagnostics on the HIGH-SPEED PRINTER. (See APPENDIX F.)

Third: Construct the generator input deck shown in FIGURE 3, selecting statements from each group in accordance with the compiler characteristics desired. A discussion of these statements appears in this appendix under the heading *Standard Version of BAC-220.*

If the HIGH-SPEED PRINTER is to be used for output at both compile time and object time, three input-output facilities statements are necessary:

```
DELETE OUTPUT FORMAT 0;
INPUTMEDIA;
OUTPUTMEDIA;
```

A machine-language deck must follow the OUTPUTMEDIA statement. This deck constitutes a subroutine which enables the compiler to produce a listing of the symbolic program on the HIGH-SPEED PRINTER, along with any syntactical error messages.

The modified library deck must have a FINISH statement both preceding and following it. The PROCESS LIBRARY statement precedes the first FINISH card, which signals the end of the generator input statements.

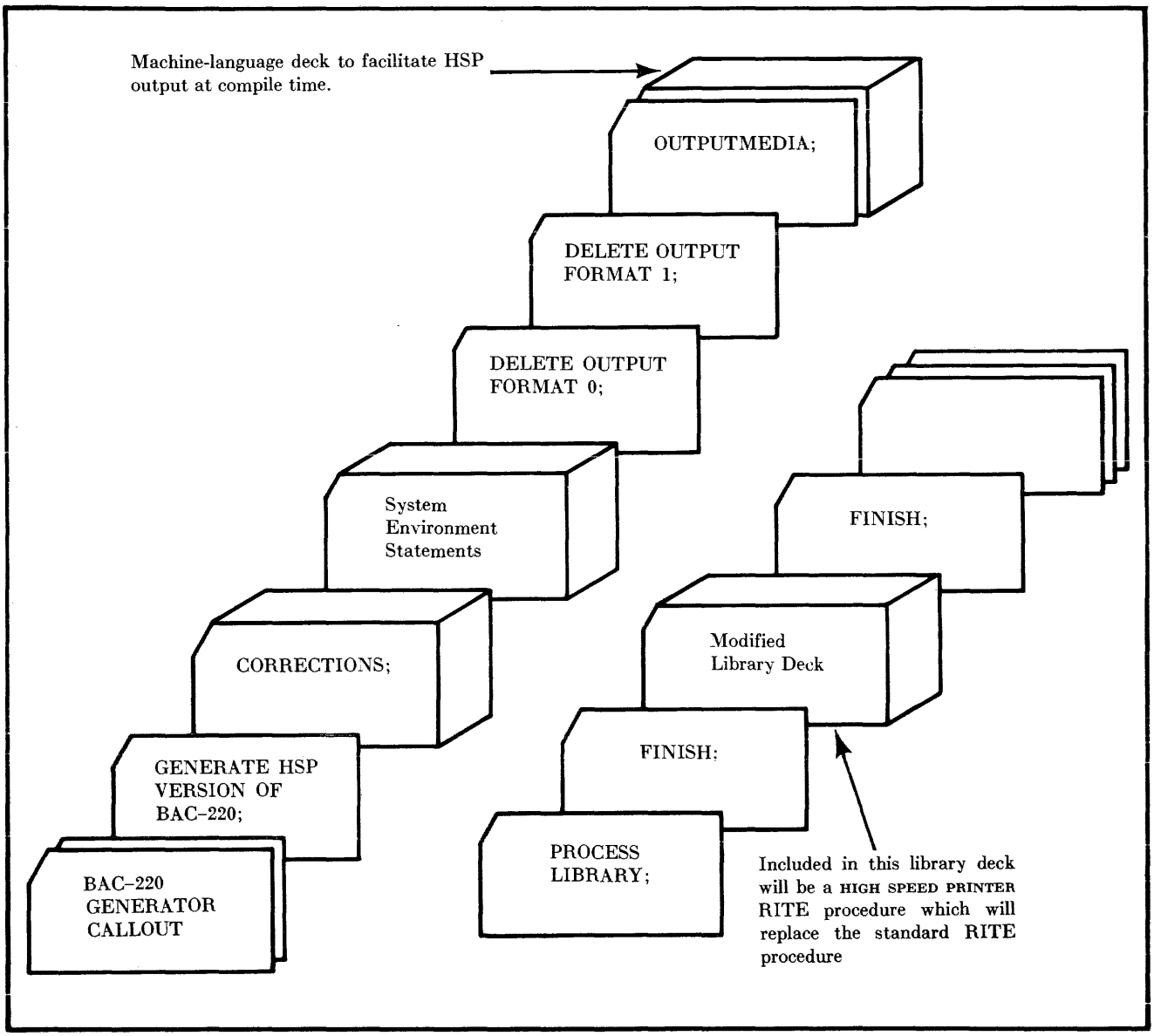


FIGURE 3. Generator Input Deck to Create HIGH-SPEED PRINTER Version (Combination 2)

The second FINISH statement delimits the modified library deck.

Paper-Tape Version of BAC-220

This version is normally required by a user whose system configuration does not include CARDATRON, or by one who wishes to use paper tape as input and obtain output from the SUPERVISORY PRINTER (see combination 3, TABLE 1). The procedure for generating this version is outlined below.

NOTE: In order to generate such a version, a CARDATRON system must be available.

First: Generate a standard version of BAC-220 to produce a standard library deck (see FIGURE 2). Note that the PUNCH LIBRARY__ INSTRUCTIONS PER CARD statement must be used.

Second: Replace the RITE procedure in the standard library deck with a special RITE procedure written by the user specifically to produce output of results, symbolic dumps, and diagnostics on the SUPERVISORY PRINTER. Replace the REED procedure in the standard library deck with a special REED procedure written specifically to allow input of data at object time through the medium of paper tape. (See APPENDIX F.)

Third: Construct the generator input deck shown in FIGURE 4, selecting statements from each group in accordance with the compiler characteristics desired. A discussion of these statements appears in this appendix under the heading *Standard Version of BAC-220.*

All input-output facilities statements must be used to generate this version. A machine-language subroutine which permits input of the source program by means of paper tape will follow the INPUTMEDIA statement. Following the OUTPUTMEDIA statement will be a machine-language subroutine which enables the compiler to provide a listing of the symbolic program, as well as syntactical error messages, on the SUPERVISORY PRINTER.

A FINISH statement follows the PROCESS LIBRARY statement. The modified library deck is placed after the FINISH statement; finally, another FINISH statement is added to complete the deck.

SPECIAL INPUT-OUTPUT ROUTINES

Several machine-language procedures and routines prepared to satisfy special input-output requirements have been published by the BURROUGHS CORPORATION in *Supplemental Material for Use with the BAC-220 Generator.* They were written to show the flexibility offered by the BAC-220 System, and are intended to encourage the user to write his own input-output procedures. For instance, the user may desire a compiler that can write the symbolic program on magnetic tape and print it

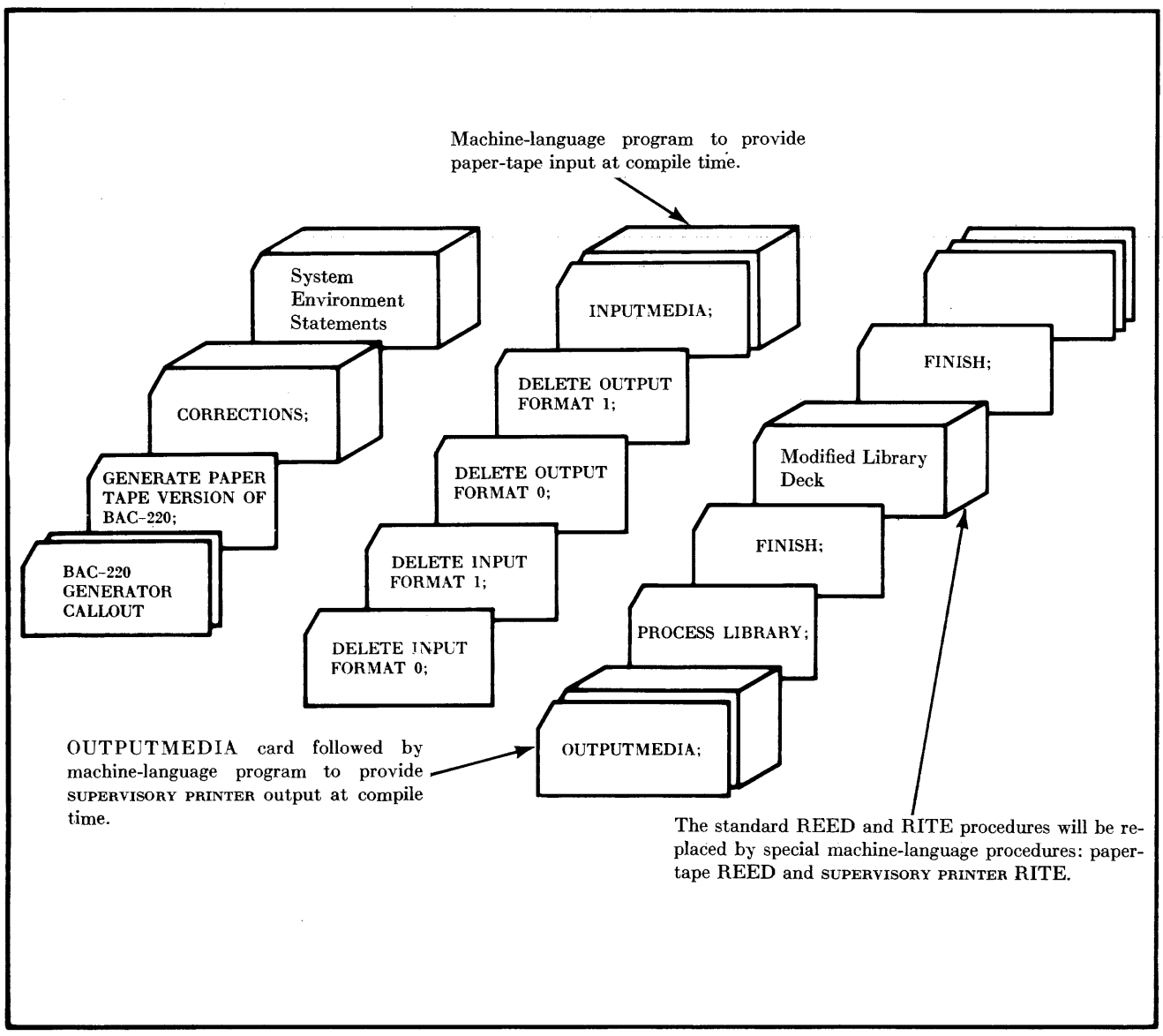


FIGURE 4. Generator Input Deck to Create Paper-Tape Version (Combination 3)

off-line on the HIGH-SPEED PRINTER through the use of a special HIGH-SPEED PRINTER plugboard. The user may write a special OUTPUTMEDIA subroutine for this purpose, one which edits the symbolic statements to suit the particular plugboard and then writes these statements on magnetic tape.

Supplemental Material for Use with the BAC-220 Generator contains the following non-standard input-output routines:

HIGH-SPEED PRINTER:

SPECIAL LIBRARY RITE PROCEDURE FOR HIGH-SPEED PRINTER AT OBJECT TIME

SPECIAL OUTPUTMEDIA SUBROUTINE FOR HIGH-SPEED PRINTER AT COMPILE TIME

PAPER-TAPE ROUTINES

SPECIAL LIBRARY REED PROCEDURE FOR PAPER TAPE AT OBJECT TIME

SPECIAL INPUTMEDIA SUBROUTINE FOR PAPER TAPE AT COMPILE TIME

SUPERVISORY PRINTER:

SPECIAL LIBRARY RITE PROCEDURE FOR SUPERVISORY PRINTER AT OBJECT TIME

SPECIAL OUTPUTMEDIA SUBROUTINE FOR SUPERVISORY PRINTER AT COMPILE TIME

(The latter procedure must be used only in conjunction with the paper-tape REED procedure listed above.)

With the exception of combination 1, at least one special-purpose input-output procedure deck must be included in the input deck to the generator program. It may be any of the following types: INPUTMEDIA, OUTPUTMEDIA, REED, or RITE. The construction of the input generator deck required for each of the input-output combinations shown in TABLE 1 is illustrated in FIGURE 1 and FIGURES 3 through 8.

ERROR MESSAGES

The generator program provides automatic recognition of error conditions. If an error condition is encountered, the computer will halt immediately after an explanation of the type of error has been emitted on the SUPERVISORY PRINTER. The following list describes the types and causes of the various error conditions which may arise prior to the library processing:

CHECK SUM ERROR

The generator program was not read in correctly. Depressing the START key causes another attempt to load the program.

ERRONEOUS CORRECTION CARD

A correction card contains invalid information.

INCORRECT STATEMENT

A symbolic statement is misspelled, or the card did not have a two-punch in column 1.

MEMORY SIZE MUST BE GIVEN AS A MULTIPLE OF ONE HUNDRED

An improper specification statement for memory size has been made.

If the library processor detects an error, it will produce one of the following error messages on the SUPERVISORY PRINTER:

EQUIVALENCE NUMBER TOO LARGE

More than two digits have been given.

IMPROPER PSEUDO-OP

The library procedure currently being processed contains an undefined pseudo-operation code.

INCORRECT PUNCTUATION

The ',' on a name card, or the '=' on an equivalence card, was replaced by some other special character, or was missing.

MISPLACED NAME CARDS

A second name card has appeared in a machine-language deck prior to the pseudo operation for FINISH which terminates that procedure.

MISSING EQUIVALENT

A machine-language instruction with a sign digit of 5 or 6 did not have an equivalence card to define its digits sL=82.

MISSING NAME CARD

The name card of the procedure to be processed is absent or does not carry a two-punch in column 1.

REFERENCE WAS MADE TO THE UNDEFINED LIBRARY PROCEDURE *name*

A machine-language procedure required as an equivalence by some other library routine was not included for processing in the modified library deck.

SEQUENCE ERROR

The relative location of the machine-language instruction being processed is less than that for the previous one.

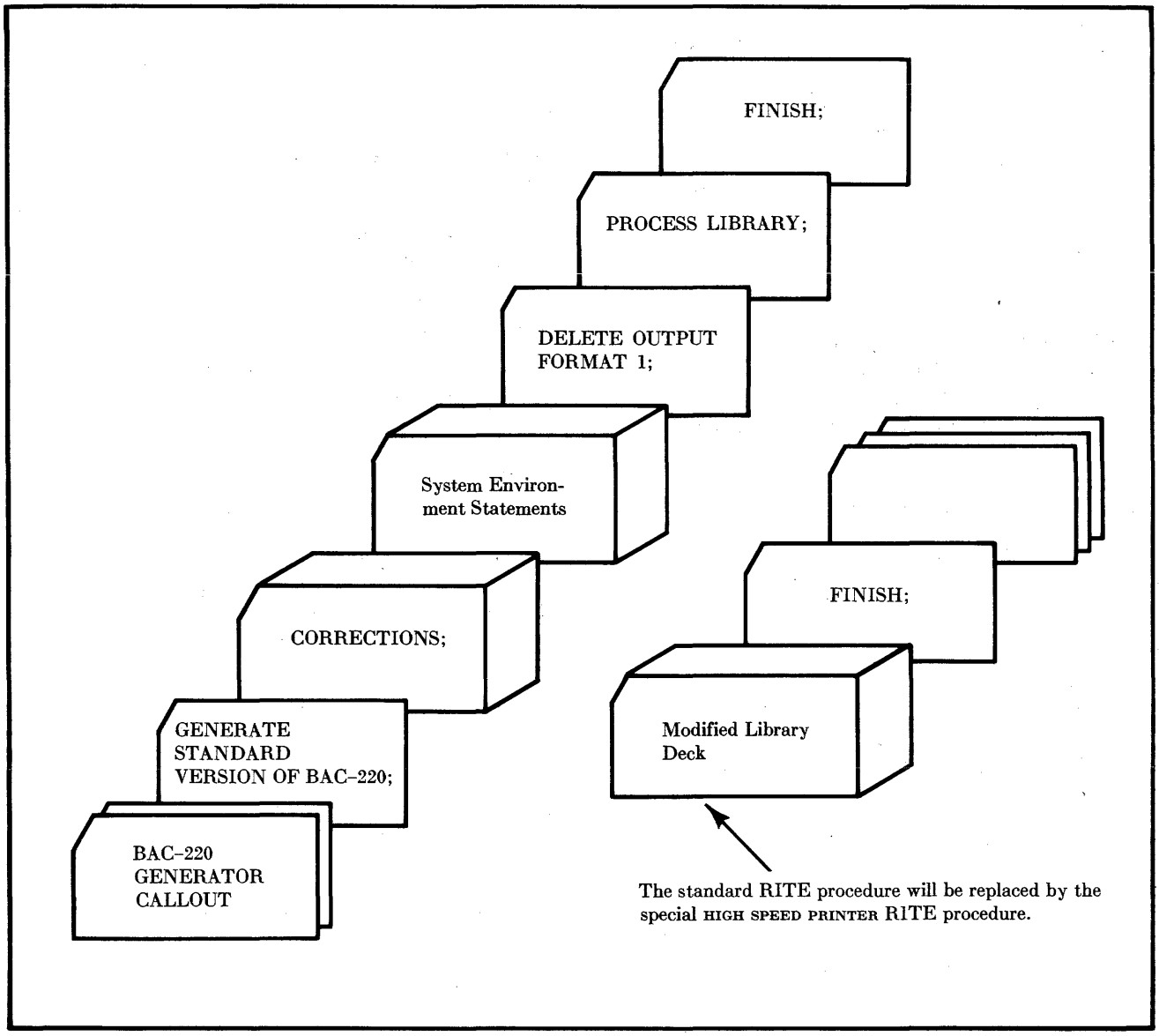


FIGURE 5. Generator Input Deck to Create Combination 4

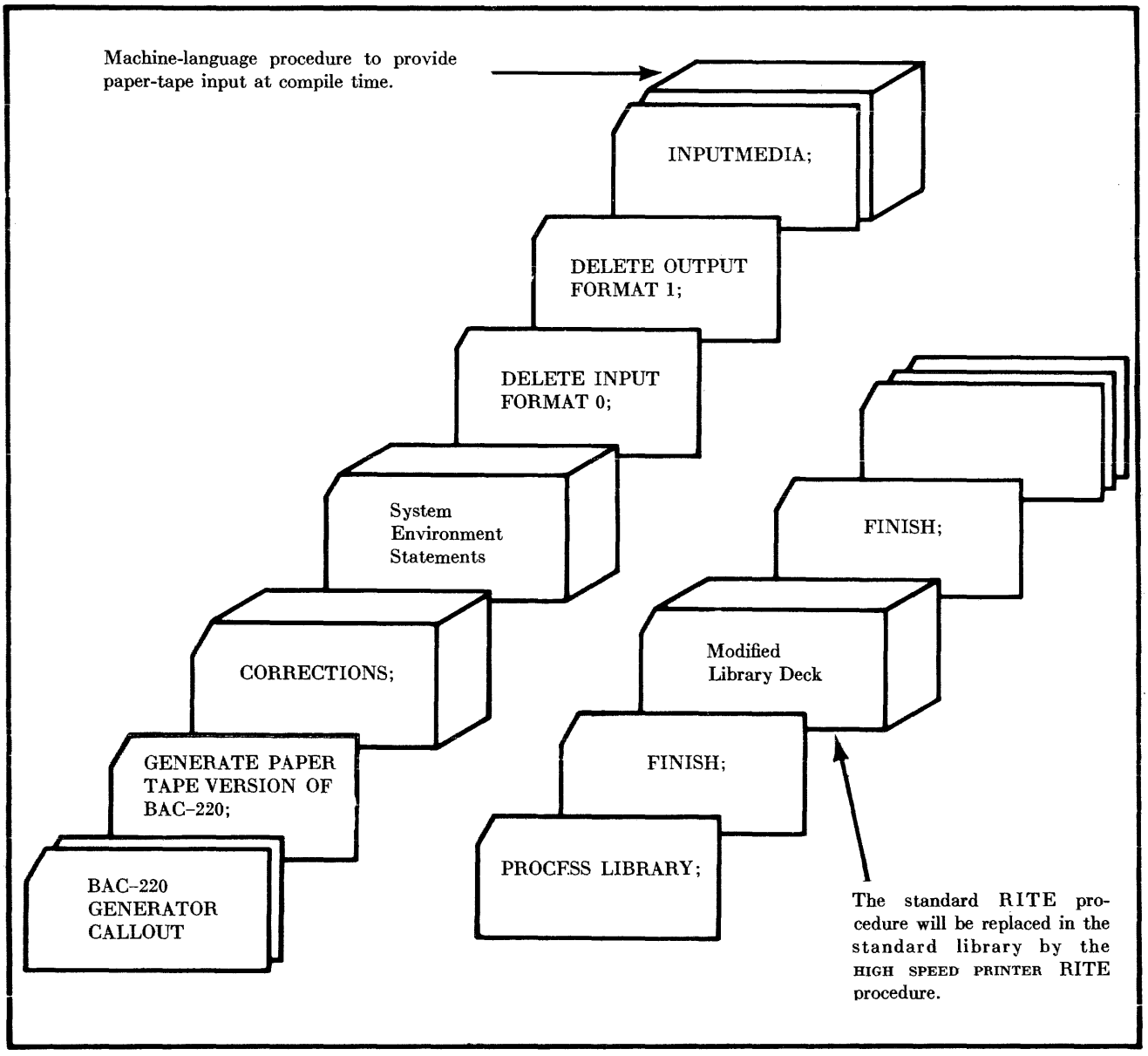


FIGURE 6. Generator Input Deck to Create Combination 5

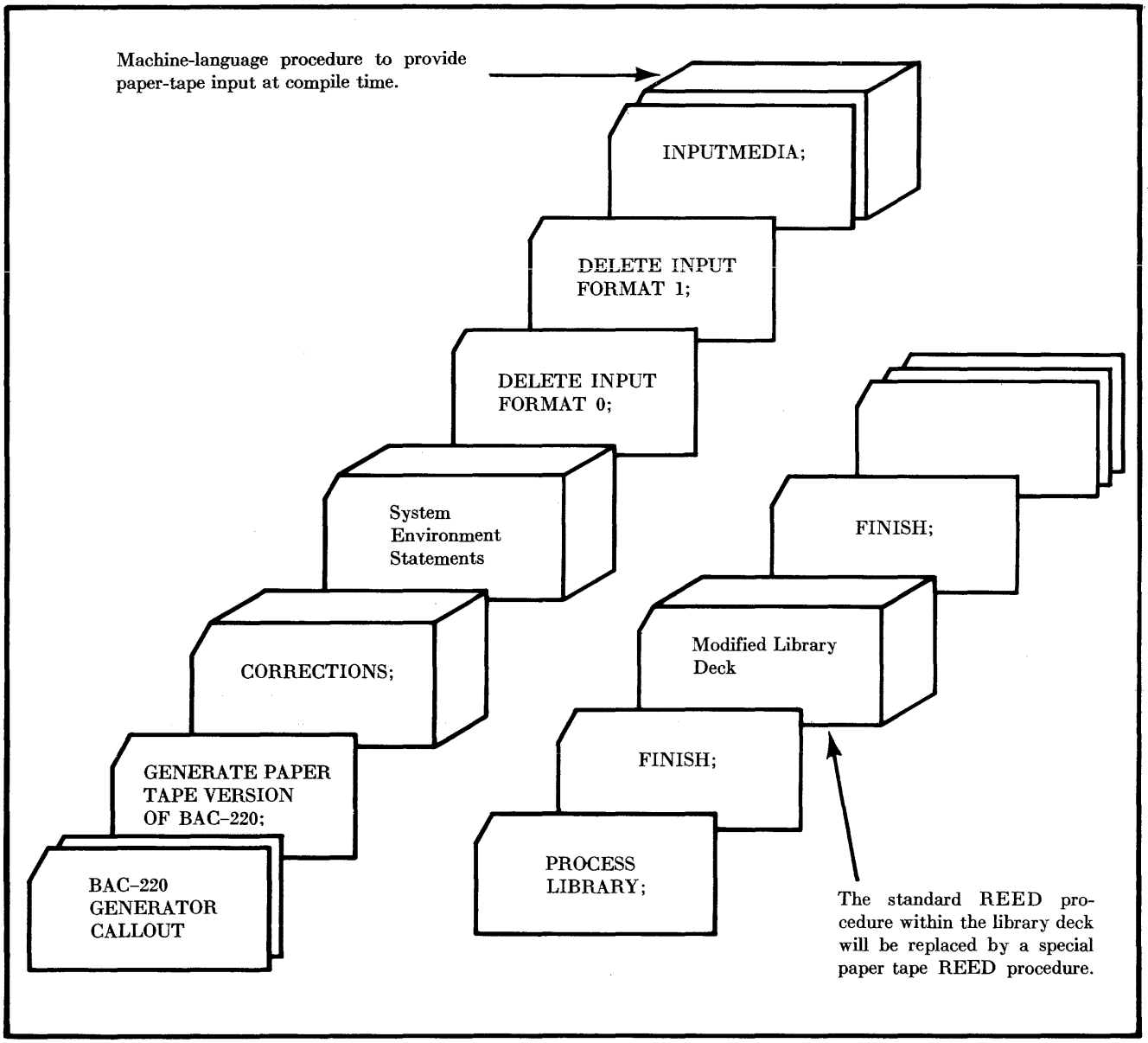


FIGURE 7. Generator Input Deck to Create Combination 6

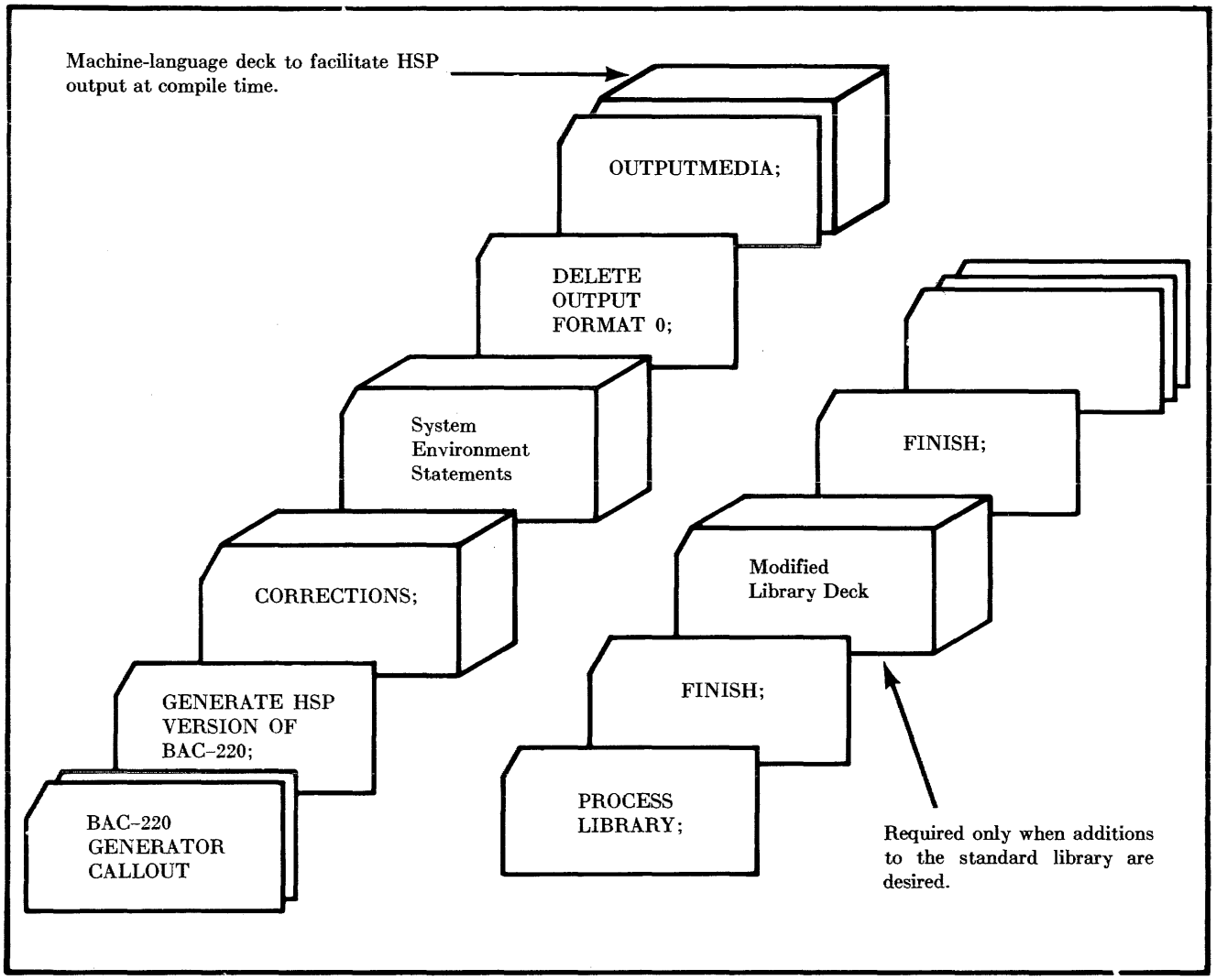


FIGURE 8. Generator Input Deck to Create Combination 7

APPENDIX B

compiler operating instructions

THIS APPENDIX CONTAINS THE INFORMATION required for the operation of the compiler. It is assumed that the user is in possession of the magnetic-tape reel containing the BAC-220 Generator and that he has generated a compiler to suit his needs.

For the most part, the compiler system is controlled by means of decks of cards which load the desired routines from the compiler tape and then transfer control to these routines. These decks are termed 'callout decks.' Any callout deck is read by placing it in the CARD READER and executing a CARD READ command: 0 *u*000 60 *xxxx*. (The address *xxxx* is irrelevant.) These decks need no blank cards preceding or following them except as specifically noted below.

The user should have access to the following callout decks: GENERATOR CALLOUT, COMPILER CALLOUT, COMPILED OBJECT PROGRAM CALLOUT, COMPILED OBJECT PROGRAM DUMP CALLOUT, and COMPILED OBJECT PROGRAM LOADER BOOTSTRAP.

PREPARATION OF SOURCE PROGRAMS ON PUNCHED CARDS

Decks containing the source-language program are punched to select format-band 2 (digit two in column 1 of each card). The statements to be compiled may occupy a predetermined field, the boundary columns of which are specified at the time of compiler generation. The standard version of the compiler scans symbolic information from column 2 through column 72.

The symbolic deck is constructed by assembling the following card deck:

First: The COMPILER CALLOUT deck;

Second: The source-language statements;

Third: External machine-language programs (if any);

Fourth: Input data (if any); and

Fifth: Three blank cards, or any number of 'reject' cards (i.e., cards with the digit seven punched in column 1).

PREPARATION OF PAPER-TAPE SOURCE PROGRAMS

Paper tape containing BAC-220 programs may be used as input to the compiler. The input routine provided by BURROUGHS scans one block of alphanumeric information at a time (14 words), and translates any special-character codes required. The first two digit positions (*sL*=22) of the first word of each block are not scanned for information. A card-to-paper-tape converter can be used for the preparation of symbolic tape if BAC-220 statements are restricted to columns 2 through 72 of the card.

The symbolic tape is prepared in the following sequence:

First: The COMPILER CALLOUT routine (below) may be punched in paper tape. This eliminates the necessity of executing the routine from the console.

```
6 u000 04 0002
0 0000 39 0000
0 cll8 50 0000
0 c001 52 0000
6 0000 30 0002
```

where *u* is the unit designation of the PAPER-TAPE READER; *c* designates the compiler storage unit; and *ll* indicates the proper lane.

Second: The BAC-220 statements must be prepared in blocks of 14 words;

Third: External machine-language programs (if any) must be prepared in blocks of 14 words;

Fourth: Input data (if any) must be prepared in blocks of 16 words (see CHAPTER VIII, *Input-Output Techniques*).

COMPILING A PROGRAM

The generated compiler is stored on a reel of magnetic tape. This tape contains the compiler program proper, the library routines, and a collection of routines which includes the symbolic dump routine, the object program loader, and the compiled object program dump. This reel is mounted on the TAPE STORAGE UNIT for which it was generated, and the unit placed in NOT-WRITE status.

A scratch tape which has been preblocked in 100-word blocks on both lanes is then mounted, the unit designator set to the TAPE STORAGE UNIT specified for machine-language output, and the unit placed in WRITE status. The symbolic deck is placed in the CARD READER and a CRD (0 *u*000 60 *xxxx*) is executed, or the symbolic tape is mounted on the PAPER-TAPE READER and a PRB (0 *u*000 04 *xxxx*) is executed. Execution of the PRB instruction assumes that the callout routine precedes the symbolic program on tape.

The compiler reads cards, paper tape, or magnetic tape, depending on its input characteristics, and produces a copy of the symbolic language on an appropriate preselected output device. Error messages resulting from compilation are indicated in the printout, and immediately precede or follow the line containing an error. As an aid in associating symbolic lines with their corresponding machine-language equivalents, a machine address is printed to the left of each symbolic card image. This address points to the location where compilation begins for the first statement on that particular card. The compiled program is written on the selected lane of the scratch tape. The opposite lane of this tape is reserved for the symbolic memory dump routine. After compilation is complete, the following two messages are produced:

COMPILED PROGRAM ENDS AT *mmmm*

PROGRAM VARIABLES START AT *nnnn*

where *mmmm* and *nnnn* are absolute addresses. The intermediate memory area between cells *mmmm* and *nnnn* is cleared.

In addition to the above cell-count messages, the A register will display either:

O K (0757 00 7250)

provided that no error messages were produced, and that this facility was not suppressed during generation of the compiler, or

X X (0525 00 5250)

in the event that the compiler detected errors.

If the compilation has been properly completed, depressing the START switch will load and execute the program.

During compilation, the following PROGRAM CONTROL SWITCHES regulate the printed output:

PCS 1 — Provides for a listing of assigned library procedures.

PCS 2 — Provides for a listing of compiled statements and constants.

PCS 3 — Provides for a listing of external program instructions.

PCS 4 — Suppresses printing of BAC-220 statements. Error messages, however, will be emitted along with a symbolic card image, which serves as an aid in locating the general vicinity of the syntactical error. If no error is discovered in the symbolic program, the object program is loaded immediately, and control is transferred to the beginning of that program.

Whenever there is a magnetic-tape malfunction, the specific type of error message is emitted, and the computer halts with 0 9669 00 9669 displayed in the C register. Depressing the START key results in another attempt to use the designated tape.

OPERATION OF THE FINISH DECLARATION

When the compiler encounters the FINISH declaration, it writes a HALT instruction (0 9669 00 9669) followed by a CARD READ or PAPER TAPE READ, BRANCH instruction at the end of the object-language program, adds the library procedures, and then stops compilation. Depressing the START key then initiates execution of the object program. Upon completion of this object-language program, pressing the START key will cause the execution of the CRD or PRB instruction as the first step in the compilation of another symbolic deck.

If the programmer wishes to avoid this sequence of events, he should precede the FINISH declaration in his symbolic program with a STOP statement, followed by a statement which transfers control back to the desired point in his program. (See CHAPTER VI.)

RELOADING THE OBJECT PROGRAM FROM MAGNETIC TAPE

The magnetic tape containing the compiled program may be remounted on its designated TAPE STORAGE UNIT at some later time, and the program loaded and run. This is accomplished by reading the deck composed of:

First: The COMPILED OBJECT PROGRAM CALLOUT deck (two cards);

Second: Input data (if any); and

Third: Three blank cards, or any number of 'reject' cards.

DUMPING A COMPILED OBJECT PROGRAM ON CARDS

After a program has been checked out, the compiled program may be punched on cards (or paper tape). This object program facility is operational with all compiled programs except those making use of the segmentation or dump features of the compiler.

Each card of the object program callout deck contains five machine-language instructions. The dump operation is complete when an OK halt appears in the C register and the A register displays all nines. At this point it is recommended that the compiled program on the output tape not be destroyed until the output produced by the CARD PUNCH is checked for errors. If a check-sum error occurs during the loading of the object program callout deck, it indicates the presence of an erroneous card, and the dump operation may have to be repeated.

A suitable loader comprising 50 cards, each card containing a sequence number in columns 11 through 14, may be punched out as an option prior to the object program itself. After the original conversion, it may be desirable to retain this loader deck for subsequent use with other compiled program decks. The loader may be dumped by setting PROGRAM CONTROL SWITCH 4 to the ON position.

Preparation for dumping the object program consists of the following:

First: Mount the compiler tape and the machine-language tape on their designated units.

Second: Place the COMPILED OBJECT PROGRAM DUMP CALLOUT deck in the CARD READER, and execute a CRD (0 *u*000 60 *xxxx*) from the console.

RELOADING THE OBJECT PROGRAM FROM CARDS

The COMPILED OBJECT PROGRAM LOADER deck is designed to be used together with the COMPILED OBJECT PROGRAM LOADER BOOTSTRAP. The combined decks constitute a complete self-loading unit for the object program. Before the object program can be loaded, the following deck must be constructed and placed in the CARD READER:

First: The COMPILED OBJECT PROGRAM LOADER BOOTSTRAP deck;

Second: The COMPILED OBJECT PROGRAM LOADER deck, which is punched with a format-digit 6 in column 1;

Third: A blank card;

Fourth: The compiled object program deck, which uses a 1-punch in column 1 for format selection;

Fifth: Input data (if any);

Sixth: Three blank cards, or any number of 'rejects.'

The loading process is begun by the execution of a CRD (0 1000 60 0000).

DUMPING A COMPILED PROGRAM ON PAPER TAPE

The object program, but not its loader, can be dumped on paper tape by depressing PROGRAM CONTROL SWITCH 3. Each object program on paper tape will be preceded by a sequence of instructions which will read in the compiled object program loader routine from the compiler tape. The procedure for loading the object program from paper tape is as follows:

First: Mount the compiler tape on the prescribed TAPE STORAGE UNIT, and designate the PAPER-TAPE READER as unit 1;

Second: Depress PROGRAM CONTROL SWITCH 3;

Third: Ready the CARD READER for input of data (if any). A blank card must precede the data deck; and

Fourth: Execute a PRB (0 1000 04 *xxxx*) from the console.

APPENDIX C

list of reserved identifiers

THE LIST BELOW INCLUDES all those identifiers to which the compiler attaches a fixed meaning. These identifiers have been mentioned separately throughout this manual, but are listed here for quick reference. A reserved identifier may not be used by the programmer for any purpose other than its function as employed by the compiler. *In addition to this list, the names of all the functions in the library should be considered as reserved identifiers.*

ABS
AND
ARRAY
BEGIN
BOOLEAN
COMMENT
DUMP
EITHER
END
ENTER
EQIV
EQL
EXTERNAL
FINISH
FLOATING
FOR
FORMAT
FUNCTION
GEQ
GO
GTR
IF
IMPL
INPUT
INTEGER
LEQ
LSS
MAX
MIN
MOD
MONITOR
NEQ
NOT
OR
OTHERWISE
OUTPUT
OVERLAY
PCS
PROCEDURE
REAL
RETURN
SEGMENT
SIGN
STATEMENT
STOP
SUBROUTINE
SWITCH
TO
TRACE
UNTIL

APPENDIX D

syntactical description of the compiler language

FORM OF DEFINITIONS

THIS APPENDIX LISTS the syntactical definitions which are provided for reference purposes. While it is not the intent to express here every rule possible for the construction of symbolic programs, these definitions should serve to answer many questions which arise concerning the language.

The definitions given here are expressed in a notation which is particularly well suited for syntactical description. A definition has the general form:

Thing being defined ::= *definition*

(the symbol ::= being read as *has the form of*).

The symbol | is to be read as *or*. Other symbols represent themselves.

For example, the definition:

⟨scale factor⟩ ::= **⟨integer constant⟩ |
**+⟨integer constant⟩ | **−⟨integer constant⟩

is to be interpreted as meaning: *A scale factor consists of two asterisks perhaps followed by a + or − sign and then followed by an integer constant.*

Syntactical definitions are recursive, that is, the definition may be applied over and over again. For example,

⟨integer constant⟩ ::= ⟨digit⟩ | ⟨integer constant⟩ ⟨digit⟩

would indicate that an integer constant consists of a string of digits.

It is impossible in these definitions to give the restrictions on the definition. For the latter, it will be necessary to refer to the relevant portion of the text, e.g., an integer constant is restricted to a maximum of ten digits (in this case, a restriction due to the word length used in the BURROUGHS 220).

BASIC SYMBOLS

⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨delimiter⟩

⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩

⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩

⟨arithmetic operator⟩ ::= + | − | . | / | * | .+ | .− | *+ | *− | /+ | /−

⟨relational operator⟩ ::= LSS | LEQ | EQL | GEQ | GTR | NEQ

⟨logical operator⟩ ::= NOT | AND | OR | EQIV | IMPL

⟨sequential operator⟩ ::= GO | GO TO | RETURN | STOP | FOR | IF | OR | EITHER IF | OR IF |
SWITCH | UNTIL | OTHERWISE | OVERLAY

⟨separator⟩ ::= . | , | .. | ; | = | ** | BEGIN | END

⟨bracket⟩ ::= (|)

⟨declarator⟩ ::= INTEGER | BOOLEAN | FLOATING | REAL | ARRAY | FUNCTION | COMMENT | PROCEDURE | SUBROUTINE | SEGMENT | FINISH | INPUT | OUTPUT | FORMAT | MONITOR | TRACE | DUMP | EXTERNAL STATEMENT | EXTERNAL PROCEDURE

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩ | ⟨identifier⟩ ⟨digit⟩

⟨number⟩ ::= ⟨unsigned number⟩ | +⟨unsigned number⟩ | −⟨unsigned number⟩

⟨unsigned number⟩ ::= ⟨integer constant⟩ | ⟨floating point constant⟩

⟨floating point constant⟩ ::= ⟨integer constant⟩ . ⟨integer constant⟩ | ⟨integer constant⟩ ⟨scale factor⟩ | ⟨integer constant⟩ . ⟨integer constant⟩ ⟨scale factor⟩

⟨scale factor⟩ ::= **⟨integer constant⟩ | **+⟨integer constant⟩ | **−⟨integer constant⟩

⟨integer constant⟩ ::= ⟨digit⟩ | ⟨integer constant⟩ ⟨digit⟩

⟨empty⟩ ::= ⟨the null string of symbols⟩

EXPRESSIONS

⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨Boolean expression⟩

⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩

⟨simple variable⟩ ::= ⟨variable identifier⟩

⟨variable identifier⟩ ::= ⟨identifier⟩

⟨subscripted variable⟩ ::= ⟨array identifier⟩ (⟨subscript list⟩)

⟨subscript list⟩ ::= ⟨subscript expression⟩ | ⟨subscript list⟩ , ⟨subscript expression⟩

⟨subscript expression⟩ ::= ⟨arithmetic expression⟩

⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ | + ⟨simple arithmetic expression⟩ | −⟨simple arithmetic expression⟩

⟨simple arithmetic expression⟩ ::= ⟨constant⟩ | ⟨variable⟩ | ⟨evaluated function⟩ | ⟨simple arithmetic expression⟩ ⟨arithmetic operator⟩ ⟨simple arithmetic expression⟩ | (⟨arithmetic expression⟩)

⟨constant⟩ ::= ⟨unsigned number⟩ | ⟨Boolean constant⟩

⟨Boolean constant⟩ ::= 0 | 1

⟨evaluated function⟩ ::= ⟨function identifier⟩ (⟨simple argument list⟩) | ⟨procedure identifier⟩ (⟨argument list⟩)

⟨function identifier⟩ ::= ⟨identifier⟩

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨simple argument list⟩ ::= ⟨simple argument⟩ | ⟨simple argument list⟩ , ⟨simple argument⟩

⟨simple argument⟩ ::= ⟨expression⟩

⟨argument list⟩ ::= ⟨input argument part⟩ | ⟨input argument part⟩ ; ⟨output argument part⟩ | ⟨input argument part⟩ ; ⟨output argument part⟩ ; ⟨program reference argument part⟩

⟨input argument part⟩ ::= ⟨empty⟩ | ⟨input argument list⟩

⟨input argument list⟩ ::= ⟨input argument⟩ | ⟨input argument list⟩ , ⟨input argument⟩

⟨input argument⟩ ::= ⟨expression⟩ | ⟨argument array⟩

⟨argument array⟩ ::= ⟨array identifier⟩ ((⟨argument subscript list⟩))

⟨argument subscript list⟩ ::= ⟨argument subscript⟩ | ⟨argument subscript list⟩ , ⟨argument subscript⟩

⟨argument subscript⟩ ::= ⟨empty⟩ | ⟨subscript expression⟩

⟨output argument part⟩ ::= ⟨empty⟩ | ⟨output argument list⟩

⟨output argument list⟩ ::= ⟨output argument⟩ | ⟨output argument list⟩ , ⟨output argument⟩

⟨output argument⟩ ::= ⟨variable⟩ | ⟨argument array⟩

⟨program reference argument part⟩ ::= ⟨program reference argument list⟩

⟨program reference argument list⟩ ::= ⟨program reference argument⟩ |
⟨program reference argument list⟩ , ⟨program reference argument⟩

⟨program reference argument⟩ ::= ⟨label⟩ | ⟨identifier⟩ ( )

⟨label⟩ ::= ⟨identifier⟩ | ⟨integer constant⟩

⟨Boolean expression⟩ ::= ⟨Boolean constant⟩ | ⟨variable⟩ | ⟨evaluated function⟩ | NOT ⟨Boolean expression⟩ |
⟨relation⟩ | ⟨Boolean expression⟩ ⟨binary logical operator⟩ ⟨Boolean expression⟩ | ((Boolean expression))

⟨relation⟩ ::= ⟨arithmetic expression⟩ ⟨relational operator⟩ ⟨arithmetic expression⟩

⟨binary logical operator⟩ ::= AND | OR | EQIV | IMPL

STATEMENTS

⟨program⟩ ::= ⟨statement body⟩; FINISH;

⟨statement body⟩ ::= ⟨statement⟩ | ⟨statement body⟩ ; ⟨statement⟩ | ⟨declaration⟩ ; ⟨statement body⟩

⟨statement⟩ ::= ⟨assignment statement⟩ | ⟨go to statement⟩ | ⟨enter statement⟩ | ⟨return statement⟩ |
⟨stop statement⟩ | ⟨switch statement⟩ | ⟨overlay statement⟩ | ⟨if statement⟩ | ⟨for statement⟩ |
⟨until statement⟩ | ⟨alternative statement⟩ | ⟨compound statement⟩ | ⟨procedure call statement⟩ |
⟨labeled dummy statement⟩ | ⟨labeled statement⟩

⟨assignment statement⟩ ::= ⟨left part list⟩ ⟨expression⟩

⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩ ⟨left part⟩

⟨left part⟩ ::= ⟨variable⟩ = ⟨procedure identifier⟩ ( ) =

⟨go to statement⟩ ::= GO TO ⟨label⟩ | GO ⟨label⟩

⟨enter statement⟩ ::= ENTER ⟨subroutine label⟩

⟨subroutine label⟩ ::= ⟨identifier⟩

⟨return statement⟩ ::= RETURN

⟨stop statement⟩ ::= STOP | STOP ⟨expression⟩

⟨switch statement⟩ ::= SWITCH ⟨expression⟩ , ((⟨switch list⟩))

⟨switch list⟩ ::= ⟨statement label⟩ | ⟨switch list⟩ , ⟨statement label⟩

⟨statement label⟩ ::= ⟨label⟩

⟨overlay statement⟩ ::= OVERLAY ⟨segment label⟩

⟨segment label⟩ ::= ⟨identifier⟩

⟨if statement⟩ ::= ⟨if clause⟩ ; ⟨statement⟩

⟨if clause⟩ ::= IF ⟨condition⟩

⟨condition⟩ ::= ⟨relation⟩ | ⟨Boolean expression⟩

⟨for statement⟩ ::= ⟨for clause⟩ ; ⟨statement⟩

⟨for clause⟩ ::= FOR ⟨variable⟩ = ⟨iteration list⟩

⟨iteration list⟩ ::= ⟨arithmetic expression⟩ | (⟨arithmetic expression⟩, ⟨arithmetic expression⟩, ⟨arithmetic expression⟩) |
⟨iteration list⟩ , ⟨iteration list⟩

⟨until statement⟩ ::= ⟨until clause⟩ ; ⟨statement⟩

⟨until clause⟩ ::= UNTIL ⟨condition⟩

⟨alternative statement⟩ ::= ⟨alternative statement head⟩ ⟨alternative statement ending⟩

⟨alternative statement head⟩ ::= EITHER ⟨if clause⟩ ; ⟨statement⟩ |
⟨alternative statement head⟩ ; OR ⟨if clause⟩ ; ⟨statement⟩

⟨alternative statement ending⟩ ::= ⟨ending⟩ | ; OTHERWISE ; ⟨statement⟩

⟨ending⟩ ::= END | END ⟨label⟩ | END ⟨procedure identifier⟩ ( )

⟨compound statement⟩ ::= BEGIN ⟨statement body⟩ ⟨ending⟩ | (⟨statement body⟩)

⟨procedure call statement⟩ ::= ⟨procedure identifier⟩ (⟨argument list⟩)

⟨dummy statement⟩ ::= ⟨empty⟩

⟨labeled statement⟩ ::= ⟨label⟩ .. ⟨statement⟩

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ | ⟨comment declaration⟩ | ⟨subroutine declaration⟩ |
⟨function declaration⟩ | ⟨procedure declaration⟩ | ⟨segment declaration⟩ | ⟨input declaration⟩ |
⟨output declaration⟩ | ⟨format declaration⟩ | ⟨monitor declaration⟩ | ⟨trace declaration⟩ | ⟨dump declaration⟩ |
⟨external declaration⟩

⟨type declaration⟩ ::= ⟨type name⟩ ⟨type list⟩ | ⟨type name⟩ OTHERWISE | ⟨type name⟩ (⟨type list⟩)

⟨type name⟩ ::= FLOATING | INTEGER | BOOLEAN | REAL

⟨type list⟩ ::= ⟨type list element⟩ | ⟨type list⟩ , ⟨type list element⟩

⟨type list element⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ ( ) | ⟨identifier⟩...

⟨array declaration⟩ ::= ARRAY ⟨array list⟩ | ARRAY (⟨array list⟩)

⟨array list⟩ ::= ⟨array list element⟩ | ⟨array list⟩ , ⟨array list element⟩

⟨array list element⟩ ::= ⟨array identifier⟩ (⟨integer list⟩) | ⟨array identifier⟩ (⟨integer list⟩) = (⟨constant list⟩)

⟨integer list⟩ ::= ⟨integer constant⟩ | ⟨integer list⟩ , ⟨integer constant⟩

⟨constant list⟩ ::= ⟨constant⟩ | ⟨constant list⟩ , ⟨constant⟩

⟨comment declaration⟩ ::= COMMENT ⟨comment symbol string⟩

⟨comment symbol string⟩ ::= ⟨any string of symbols not containing ';'⟩

⟨subroutine declaration⟩ ::= SUBROUTINE ⟨subroutine label⟩ ; ⟨compound statement⟩

⟨function declaration⟩ ::= FUNCTION ⟨function identifier⟩ (⟨simple parameter list⟩) = ⟨expression⟩

⟨simple parameter list⟩ ::= ⟨identifier⟩ | ⟨simple parameter list⟩ , ⟨identifier⟩

⟨procedure declaration⟩ ::= PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩) ; ⟨compound statement⟩

⟨parameter list⟩ ::= ⟨input parameter part⟩ | ⟨input parameter part⟩ ; ⟨output parameter part⟩ |
⟨input parameter part⟩ ; ⟨output parameter part⟩ ; ⟨program reference parameter part⟩

⟨input parameter part⟩ ::= ⟨empty⟩ | ⟨input parameter list⟩

⟨input parameter list⟩ ::= ⟨input parameter⟩ | ⟨input parameter list⟩ , ⟨input parameter⟩

⟨input parameter⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ (⟨empty subscript list⟩)

⟨empty subscript list⟩ ::= ⟨empty⟩ | ⟨empty subscript list⟩ , ⟨empty⟩

⟨output parameter part⟩ ::= ⟨empty⟩ | ⟨output parameter list⟩

⟨output parameter list⟩ ::= ⟨output parameter⟩ | ⟨output parameter list⟩ , ⟨output parameter⟩

⟨output parameter⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ (⟨empty subscript list⟩)

⟨program reference parameter part⟩ ::= ⟨program reference parameter list⟩

⟨program reference parameter list⟩ ::= ⟨program reference parameter⟩ |
⟨program reference parameter list⟩ , ⟨program reference parameter⟩

⟨program reference parameter⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ ( )

⟨segment declaration⟩ ::= SEGMENT ⟨segment label⟩ ; ⟨compound statement⟩

⟨input declaration⟩ ::= INPUT ⟨input list⟩ | INPUT (⟨input list⟩)

⟨input list⟩ ::= ⟨input list element⟩ | ⟨input list⟩ , ⟨input list element⟩

⟨input list element⟩ ::= ⟨input label⟩ (⟨input data list⟩)

⟨input data list⟩ ::= ⟨input data list element⟩ | ⟨input data list⟩ , ⟨input data list element⟩

⟨input label⟩ ::= ⟨identifier⟩

⟨input data list element⟩ ::= ⟨variable⟩ | ⟨for clause⟩ ; ⟨input data list element⟩ |
⟨for clause⟩ ; (⟨input data list⟩)

⟨output declaration⟩ ::= OUTPUT ⟨output list⟩ | OUTPUT (⟨output list⟩)

⟨output list⟩ ::= ⟨output list element⟩ | ⟨output list⟩ , ⟨output list element⟩

⟨output list element⟩ ::= ⟨output label⟩ (⟨output data list⟩)

⟨output data list⟩ ::= ⟨output data list element⟩ | ⟨output data list⟩ , ⟨output data list element⟩

⟨output label⟩ ::= ⟨identifier⟩

⟨output data list element⟩ ::= ⟨expression⟩ | ⟨for clause⟩ ; ⟨output data list element⟩ | ⟨for clause⟩ ; (⟨output data list⟩)

⟨format declaration⟩ ::= FORMAT ⟨format list⟩ | FORMAT (⟨format list⟩)

⟨format list⟩ ::= ⟨format list element⟩ | ⟨format list⟩, ⟨format list element⟩

⟨format list element⟩ ::= ⟨format label⟩ (⟨format data list⟩)

⟨format data list⟩ ::= ⟨format data list element⟩ | ⟨format data list⟩ , ⟨format data list element⟩ |
(⟨format data list⟩)

⟨format label⟩ ::= ⟨identifier⟩

⟨format data list element⟩ ::= *⟨format string⟩* | ⟨letter⟩ ⟨integer constant⟩ |
⟨letter⟩ ⟨integer constant⟩ . ⟨integer constant⟩ | ⟨repeat phrase⟩ | ⟨activation phrase⟩

⟨activation phrase⟩ ::= ⟨letter⟩ | ⟨letter⟩ ⟨integer constant⟩ | ⟨integer constant⟩ ⟨activation phrase⟩

⟨repeat phrase⟩ ::= ⟨integer constant⟩ ⟨format data list⟩

⟨format string⟩ ::= ⟨any string of symbols not containing '*'⟩

⟨monitor declaration⟩ ::= MONITOR ⟨monitor list part⟩

⟨monitor list part⟩ ::= ⟨empty⟩ | ⟨monitor list⟩

⟨monitor list⟩ ::= ⟨monitor list element⟩ | ⟨monitor list⟩ , ⟨monitor list element⟩

⟨monitor list element⟩ ::= ⟨identifier⟩ | ⟨label⟩

⟨trace declaration⟩ ::= TRACE ⟨trace list⟩

⟨trace list⟩ ::= ⟨trace list element⟩ | ⟨trace list⟩ , ⟨trace list element⟩

⟨trace list element⟩ ::= ⟨label⟩ | ⟨label⟩ (⟨integer constant⟩)

⟨dump declaration⟩ ::= DUMP ⟨dump list part⟩

⟨dump list part⟩ ::= ⟨empty⟩ | ⟨dump list⟩

⟨dump list⟩ ::= ⟨dump list element⟩ | ⟨dump list⟩ , ⟨dump list element⟩

⟨dump list element⟩ ::= ⟨identifier⟩ | ⟨label⟩

⟨external declaration⟩ ::= EXTERNAL STATEMENT ⟨label⟩ |
EXTERNAL PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩) |
EXTERNAL PROCEDURE ⟨procedure identifier⟩ (⟨parameter list⟩) ; ⟨type name⟩ ⟨procedure identifier⟩

APPENDIX E

transliteration rules

THIS APPENDIX presents a summary of equivalences between the elements of the BURROUGHS Algebraic Compiler language and the elements of ALGOL, the international algebraic language. It is this latter language which is used for the publication of programs written in the former. Transliteration of programs for the BURROUGHS Algebraic Compiler into ALGOL requires a thorough familiarity with ALGOL, the details of which are available in the literature.†

† See *Communications of the ACM*, vol. 1, no. 12, pp. 8-22; and vol. 3, no. 5, pp. 299-313.

1. BASIC SYMBOLS

	Reference Language	Burroughs Language
a. *Non-delimiters*		
(1) Letters	A ⎓ Z	A ⎓ Z
	a ⎓ z	A ⎓ Z
(2) Digits	0 ⎓ 9	0 ⎓ 9
b. *Delimiters*		
(1) Operators		
Arithmetic	+	+
	−	−
	×	. The multiplication sign may be omitted in certain instances. It is represented on card equipment as a decimal point (.).
	/	/
Relational	<	LSS
	≤	LEQ
	=	EQL
	≥	GEQ
	>	GTR
	≠	NEQ

1. BASIC SYMBOLS (continued)

b. *Delimiters* (continued)

(1) Operators (continued)

Logical	¬	NOT
	∨	OR
	∧	AND
	≡	EQIV
	(⊃)	IMPL
	Logical implication — no equivalent in the reference language.	
Sequential	*go to*	GO TO
	(no equivalent)	SWITCH
	do	(no equivalent)
	return	RETURN
	stop	STOP
	for	FOR
	if	IF
	or	OR
	if either	EITHER IF
	or if	OR IF
	(no equivalent)	UNTIL
(2) Separators	(not required)	⟨space⟩
		Spaces must be used to separate contiguous identifiers or an identifier followed by a constant. *Spaces may not be imbedded within an identifier or a constant.*
	.	.
	,	,
	:	..
	;	;
		On standard keypunch equipment, the semicolon is represented by $ (dollar sign).
	:=	=
	=:	(no equivalent)
	→	(no equivalent)
		Related to the DO statement in the reference language.
	10	**
		Power of 10.
(3) Brackets	*begin*	BEGIN
	end	END
	(	(
	)	)
	[	(
	]	)

1. BASIC SYMBOLS (continued)

b. *Delimiters* (continued)

(3) Brackets (continued)

↑	*(
↓	)
	(Parentheses may be omitted in certain instances.)

(4) Declarators

type	INTEGER, BOOLEAN, FLOATING, REAL
array	ARRAY
(no equivalent)	FUNCTION
comment	COMMENT
procedure	PROCEDURE
(no equivalent)	SUBROUTINE
(no equivalent)	INPUT
(no equivalent)	OUTPUT
(no equivalent)	FORMAT
(no equivalent)	FINISH
switch	(no equivalent) See SWITCH statement.

2. EXPRESSIONS

Most expressions are self-evident, except those noted below:

a. Numbers	$G.G_{10} \pm G$	$\mathcal{G}.\mathcal{G}^{**} \pm \mathcal{G}$
	G.	$\mathcal{G}.0$
	.G	$0.\mathcal{G}$
	10^G	$1^{**}\mathcal{G}$
b. Simple Variables	I	$\mathcal{I}$
c. Subscripted Variables	I[C]	$\mathcal{I}(\mathcal{C})$
d. Functions	I(R)	$\mathcal{I}(\mathcal{R})$
e. Arithmetic Expressions	(*See* Arithmetic Operators, *above.*)	
	$E_1 \uparrow E_2 \downarrow$	E1*(E2) (Parentheses may be omitted in certain instances.)
f. Boolean Expressions	(*See* Relational Operators, *above.*)	

3. STATEMENTS

a. Assignment	V := E	$\mathcal{V} = \mathcal{E}$
b. Go to	*go to* D	GO TO D
c. Switch	(no equivalent)	SWITCH $\mathcal{E}$, ($\mathfrak{N}_1$, $\mathfrak{N}_2$, ⌣, $\mathfrak{N}_n$)
d. If	*if* B	IF $\mathcal{B}$
e. Until	(no equivalent)	UNTIL $\mathcal{C}$

3. STATEMENTS (continued)

f. For	*for* V := C	FOR V = $\mathfrak{C}$
	for V := E_{i_1} (E_{s_1}) E_{e_1}, ⁓	FOR V = ($\mathcal{E}_{i_1}$, $\mathcal{E}_{s_1}$, $\mathcal{E}_{e_1}$)
g. Alternative	*if either* B_1; Σ_1; *or if* B_2; Σ_2; ⁓ *end*	EITHER IF $\mathfrak{B}_1$; $\mathcal{S}_1$; OR IF $\mathfrak{B}_2$; $\mathcal{S}_2$ END EITHER IF $\mathfrak{B}_1$; $\mathcal{S}_1$; OR IF $\mathfrak{B}_2$; $\mathcal{S}_2$; OTHERWISE; $\mathcal{S}_n$
h. Stop	*stop*	STOP
i. Return	*return*	RETURN
j. Procedure	I(P_i, P_i, ⁓, P_i) =: (P_o, P_o, ⁓, P_o)	$\mathcal{G}$($\mathcal{P}_i$, $\mathcal{P}_i$, ⁓, $\mathcal{P}_i$; $\mathcal{P}_o$, $\mathcal{P}_o$, ⁓, $\mathcal{P}_o$; $\mathcal{P}_r$, $\mathcal{P}_r$, ⁓, $\mathcal{P}_r$)
k. Subroutines	(no equivalent)	ENTER $\mathcal{G}$
l. Do	*do* L_1, L_2 ($S_\rightarrow \rightarrow$ I, ⁓, $S_\rightarrow \rightarrow$ I)	(no equivalent)

4. DECLARATIONS

a. Type	*type* (I, I, ⁓, I)	BOOLEAN $\mathcal{G}$, $\mathcal{G}$(,), $\mathcal{G}$... INTEGER $\mathcal{G}$, $\mathcal{G}$(,), $\mathcal{G}$... FLOATING $\mathcal{G}$, $\mathcal{G}$(,), $\mathcal{G}$... REAL $\mathcal{G}$, $\mathcal{G}$(,), $\mathcal{G}$...
	(no equivalent)	⟨type⟩ OTHERWISE
b. Array	*array* (I, I, ⁓, I[C:C′], I. I, ⁓)	ARRAY $\mathcal{G}_i$ ($\mathfrak{C}_i$),
	(no equivalent)	ARRAY $\mathcal{G}$; ($\mathfrak{D}_i$) = ($\mathfrak{C}_i$),
c. Functions	I(R) := E	FUNCTION $\mathcal{G}$($\mathfrak{R}$) = $\mathcal{E}$
d. Comment	*comment* S;	COMMENT $\mathcal{S}$;
e. Procedure	*procedure* I(P_i) =: (P_o), I(P_i) =: (P_o), ⁓⁓⁓, I(P_i) =: (P_o) Δ; Δ; ⁓⁓⁓; Δ *begin* Σ; Σ; ⁓⁓⁓; Δ; Δ; ⁓⁓⁓; Σ; Σ *end*	PROCEDURE $\mathcal{G}$($\mathcal{P}_i$; $\mathcal{P}_o$; $\mathcal{P}_r$); BEGIN $\mathcal{S}_i$ END
f. Subroutines	(no equivalent)	SUBROUTINE $\mathcal{G}$; ($\mathcal{S}_i$)
g. Input	(no equivalent)	INPUT ($\mathcal{G}_i$ ($\mathfrak{DL}$)$_i$)
h. Output	(no equivalent)	OUTPUT ($\mathcal{G}_i$ ($\mathfrak{DL}$)$_i$)
i. Format	(no equivalent)	FORMAT ($\mathcal{G}_i$ ($\mathfrak{FDL}$)$_i$)
j. Finish	(no equivalent)	FINISH
k. Diagnostic	(no equivalent)	MONITOR $\mathfrak{ML}$
	(no equivalent)	TRACE $\mathfrak{IL}$
	(no equivalent)	DUMP $\mathfrak{DL}$

APPENDIX F

construction of machine-language programs

PROGRAMS WRITTEN IN MACHINE-LANGUAGE—whether for use as external programs, as external statements, or as library procedures—are prepared similarly, and the techniques and conventions required for their construction are listed below.

LINKAGE TO PROCEDURES

When the compiler provides linkage to any procedure, the instructions assembled are of the form:

0000 STP *aaaa*
*nn*00 BUN *aaaa*

where *aaaa* is the location of the first cell of the procedure and the value of *nn* is one less than the number of parameters to be given to the procedure.

Since the address fields of both the STP and the BUN instructions are the same, it is necessary that the first instruction of the procedure be a NOP. This NOP is of the form:

aaaa: *bbbb* NOP *xxxx*

where *xxxx* will be replaced when the STP instruction is executed, and *bbbb* is the address in which the first parameter is to be stored. Each succeeding parameter will then be stored in *descending* sequence beginning with the address *bbbb*.

The last parameter is always retained in the A register and will not be placed in memory. In all cases if a procedure has only one parameter, it will be found in the A register.

Suppose that P_1, P_2, and ..., P_m are the addresses of those parameters which are to be given to the procedure. The compiler will in effect produce the following coding when the procedure-call statement is encountered:

CODE			REMARKS
0	0000	CAD P_1	
0	4400	DLB *aaaa*	*bbbb* → rB
1	0000	STA 0000	P_1 → *bbbb*
0	0000	CAD P_2	
1	0000	STA 9999	P_2 → *bbbb* − 1
		.	.
		.	.
		.	.
0	0000	CAD P_{m-1}	
1	0000	STA 9999−*m*+2	P_{m-1} → *bbbb* − *m* + 2
0	0000	CAD P_m	P_m → rA
0	0000	STP *aaaa*	
0	*nn*00	BUN *aaaa*	(*nn* = *m* − 1)

Notice that the control field of the NOP instruction which heads the procedure provides the address used to determine the location in which to store parameters.

When constructing machine-language procedures, the *bbbb* field may be either located within the procedure code itself (the control field of an instruction may be relocated—see *Relocation Conventions*) or it may be absolute. The absolute addresses available for this purpose are 0100 through 0199. A word of caution concerning the use of any of these absolute addresses for *bbbb* is necessary. The memory area 0100 through 0199 is used as a buffer area by input-output procedures.

PARAMETERS OF PROCEDURES

As discussed in CHAPTER VII, parameters may be considered as being either values or identifiers. Input variables or expressions are values; the procedure thus receives the actual value of the variable or expression given as a parameter. Output variables are identifiers.

When an output variable is indicated, the address of that variable is given to the procedure in the 04-field.

All program-reference parameters are also identifiers; the procedure receives, in the 04-field, the address of the parameter to which reference is made.

In the case of arrays the situation is somewhat more complex. Whenever an array is written as a parameter (either as input or output) provision is made for several parameters to be given to the procedure. This set of parameters consists of a base address and values corresponding to each empty subscript position, which we will call m, μ_1, μ_2, ..., μ_k, respectively. The address of an element A(n_1, n_2, ..., n_k) is then given by

$$m + (((...(n_1\mu_1 + n_2)\mu_2 + n_3) ... + n_k)\mu_k.$$

As an example, consider the three-dimensional array M(, , 5, ,7) which is used as an argument for a given procedure. The arguments supplied to this procedure would be m, μ_1, μ_2 and μ_3. The procedure could then calculate the address of the element M(n_1, n_2, 5, n_3, 7) by

$$m + ((n_1\mu_1 + n_2)\mu_2 + n_3)\mu_3.$$

It should be noted that μ_k represents the spacing between adjacent elements in a row of the array. In the common case where all subscripts are empty, or in the special case where all are specified except the rightmost subscript, the value of μ_k will be *one*. The programmer may make use of this fact to eliminate the final multiplication if he can guarantee that his external program will always be utilized with these restrictions. He must, however, allow space in his parameter buffer area for μ_k, since the compiler will always supply it.

EXAMPLE:

Suppose the declaration EXTERNAL PROCEDURE M(A(,,), B) appeared in the symbolic program, and it is necessary to refer to the element A(I,J,K,) from inside the external program. The following coding would place the value of the required element in the A register:

LOCATION	OPERATION AND ADDRESS	REMARKS
0000	7 0125 01 0000	NOP 0000
0001	8 0000 40 0121	STA B
.	.	Parameter buffer is specified as cell 0125 relative to this instruction.
.	.	
.	.	
.	.	
0035	8 0000 10 0126	CAD I
0036	8 0000 14 0124	MUL μ_1
0037	8 0001 49 0010	SLT 10
0038	8 0000 12 0127	ADD J
0039	8 0000 14 0123	MUL μ_2
0040	0 0001 49 0010	SLT 10
0041	8 0000 12 0128	ADD K
0042	8 0000 14 0122	MUL μ_3
0043	8 0410 40 0045	STR 0045
0044	8 0000 42 0125	LDB m
0045	1 0000 10 0000	CAD array element
.	.	.
.	.	.
.	.	.
0121	0 0000 00 0000	B
0122	0 0000 00 0000	μ_3
0123	0 0000 00 0000	μ_2
0124	0 0000 00 0000	μ_1
0125	0 0000 00 0000	m base address of array A
0126	0 0000 00 0003	I
0127	0 0000 00 0007	J
0128	0 0000 00 0006	K

RELOCATION CONVENTIONS

All machine-language programs are written relative to location 0000. The compiler will relocate the program so that it occupies storage starting at some other cell. The address of this cell is called the *relocation base*. This relocation is controlled by the sign digit of the machine-language commands.

Sign Digit of Zero, One, Two, or Three

Instructions with signs of zero, one, two, or three are not altered in any way.

Sign Digit of Four

Instructions with a sign digit of four behave in a manner analogous to pseudo-operation codes in an assembler. The following pseudo-operations are allowed:

An instruction with a sign of four and an operation code of 00 (sL=62) is used to reserve blocks of memory relative to the beginning of the external program. The location counter is advanced by the amount specified in the address field of the pseudo-operation instruction.

If an instruction has a sign of four and an operation code of 01, it means that the *next* instruction or word will be an 11-digit constant.

An instruction with a sign of four and an operation code of 02 indicates that the *next* instruction will be a CARDATRON input-output instruction, and that the unit designations should be changed by the library processor to the correct ones specified by the system environment statements of the generator. Thus if the next instruction is a CWR with a unit designation of 1, it will be changed to the unit specified for the CARD PUNCH; if the

unit designation is 2, it will be changed to the unit assigned to the printer.

A machine-language instruction with a sign of four and an operation code of 03 enables the address field of the *next* instruction to be modified by the addition of a four-digit constant to the address field of the pseudo-instruction. Whenever this pseudo-operation precedes a machine instruction having a sign of five or six, the four-digit constant will increase the address supplied to the instruction through the use of an equivalence card. As a result, it is possible to refer to any location relative to the address assigned to the identifier used in the equivalence.

An instruction with a sign of four and an operation code of 04 permits the control field of the *next* instruction to be relocated with respect to the first instruction of the external program. It is commonly used to precede an instruction which has a sign digit of five or six.

If the operation code is 30, the compiler will insert an unconditional transfer to the statement immediately following the declarator which defined the external statement. This instruction is similar to the RETURN operation of the symbolic language and may be used any number of times within an external statement.

If the operation code is 99, the end of this machine-language program is indicated.

Sign Digit of Five or Six

Instructions with a sign digit of five or six have their address field located relative to some identifier defined within the ALGOL program. The control field is unaltered. The sign is set to one if the original sign was five, and to zero if it was six. (See *Use of Equivalence Cards* in this appendix.)

Sign Digit of Seven

Instructions with a sign digit of seven have their control field (sL=44) relocated with respect to the first instruction of this external program. The address field is not altered and the sign of the instruction is set to zero.

An instruction with a sign of seven and an operation code of 01 (NOP) is often used as the first instruction of an external or library procedure when it is desired to locate the parameters to the procedure within the routine itself.

Sign Digit of Eight

Instructions with a sign digit of eight have their address fields (sL=04) relocated with respect to the first instruction of this external program. The control field is not altered and the sign of the instruction is set to zero.

Sign Digit of Nine

Instructions with a sign digit of nine have their address fields relocated with respect to the first instruction of this external program. The control field is not altered and the sign of the instruction is set to one.

MAGNETIC-TAPE OPERATIONS

In order to allow the use of the MAGNETIC-TAPE FIELD SEARCH (MFS) and MAGNETIC-TAPE FIELD SCAN (MFC) commands, the following conventions have been employed:

The pseudo-operation codes 90 and 91 have been introduced for use by the programmer, when referring to the MFS and MFC commands, respectively.

If the address field of the field search or scan is to be absolute, then the sign of the instruction must be four or five.

If the address field is relative to the first line of the subroutine, the sign must be eight or nine.

If the address field is relative to some identifier, the sign must be six or seven.

The signs of five, seven, and nine indicate B-modification is to be performed.

USE OF EQUIVALENCE CARDS

It is possible in a machine-language program to refer to any identifier defined within the symbolic program, as well as to any library procedure. Every identifier to which it is desired to refer is assigned a unique two-digit equivalent, *mm*, by means of an equivalence card preceding the machine-language deck. (See APPENDIX G for the list of subroutine names.) This card has the digit two in column 1, an arbitrary number of spaces, an identifier, an equal sign, and a two-digit number (leading zeros may be omitted) which is assigned to the identifier, e.g.:

COLUMNS	CONTENTS
1	2
5–9	SIN = 3
or:	
1	2
7–19	SUMSQUARES = 56

If the identifier is defined within a procedure, *then it must be preceded by a prefix which is the name of the procedure enclosed in parentheses.* Thus the equivalence card

for label START within the procedure SIMPSON might appear as:

COLUMNS	CONTENTS
1	2
4–20	(SIMPSON)START = 13

The address field of any instruction within a machine-language program which has a sign of five or six, i.e.:

$\left\{\begin{matrix}5\\6\end{matrix}\right\}$ *xxxx* OP *mm kk*

will be replaced by the constant *kk*, plus the address of the identifier corresponding to the equivalent *mm*.

For example, suppose that the instructions

```
6 0000 44 0300
6 0001 30 0301
5 0001 23 0341
```

are included within a machine-language program and that the equivalence card with the following entries:

COLUMNS	CONTENTS
1	2
4–8	SIN=3

has preceded this program. Now assume that the compiler has assigned the cells 2856 - 2904 to the SIN routine. These instructions would then appear in the final program as:

```
0 0000 44 2856
0 0001 30 2857
1 0001 23 2897
```

If the identifier used in the equivalence statement is that of an array, special coding must be provided to address an element within this array. The method assumes that an array A has been defined previously by the declaration:

ARRAY A(μ_1, μ_2, ..., μ_k)

If this is so, then the address of the element A(n_1, n_2, ..., n_k) is given by $m + (((...(n_1 \mu_2 + n_2)\mu_3 + n_3) ... + n_{k-1})\mu_k + n_k$. (Note that μ_1 is not used in the calculation.) It should be noted that this formula differs from that given in *Parameters of Procedures*. When referring to an array *as declared*, the elements are adjacent in memory; in consequence, the final multiplication is eliminated.

To each array identifier the compiler assigns a value of *m*, representing the base address of that array. This address will then replace the equivalence number assigned to the identifier at the head of the external program. Thus if the array G had been defined by the declaration:

ARRAY G(15, 4, 6)

and it is required to refer to the element G(n_1, n_2, n_3) from inside of an external program by means of an equivalence statement, then the following special section of coding must be included to place the value of the element into the A register.

RELATIVE LOCATION	OPERATION AND ADDRESS	REMARKS	
.	.	.	.
0107	8 0000 10 0132	CAD	n_1
0108	8 0000 14 0130	MUL	μ_2
0109	0 0001 49 0010	SLT	10
0110	8 0000 12 0133	ADD	n_2
0111	8 0000 14 0131	MUL	μ_3
0112	0 0001 49 0010	SLT	10
0113	8 0000 12 0134	ADD	n_3
0114	8 0000 40 0135	STA	temp
0115	8 0000 42 0135	LDB	temp
0116	5 0000 10 5900	– CAD	*m*
.	.		.
.	.		.
.	.		.
0130	0 0000 00 0004		μ_2
0131	0 0000 00 0006		μ_3
0132	0 0000 00 (n_1)		n_1
0133	0 0000 00 (n_2)		n_2
0134	0 0000 00 (n_3)		n_3
0135	0 0000 00 0000		temp

The external program making use of the above coding must be preceded by an equivalence card with the format:

COLUMNS	CONTENTS
1	2
6–9	G=59

DESCRIPTION OF NAME CARDS

A 'name' (or header) card must precede each external program in order to relate it to the declaration in the source program.

GENERAL FORM:

$\mathcal{g}$, $\mathfrak{FJ}$

where $\mathcal{g}$ is an identifier declared to be the name of the external procedure or statement;

$\mathfrak{FJ}$ is the function type, where an external procedure defines a value, and may be any one of the reserved

words FLOATING, REAL, BOOLEAN or INTEGER. If the external program is an EXTERNAL statement, or an EXTERNAL procedure which does not define a value, then the type designation ƒƒ may be omitted.

PREPARATION OF EXTERNAL PROGRAMS

As discussed in CHAPTER VII, it is necessary, when using external programs, to have an EXTERNAL declarator in the body of the symbolic program. The definition of the program in BURROUGHS 220 machine-language follows the FINISH declaration of the symbolic program.

All the external programs declared in the symbolic program follow one after the other. Following the final external program, the word FINISH must appear on another symbolic card. This defines the end of the program (symbolic statements and machine-language) to the compiler; it is in addition to the FINISH declaration of the symbolic program.

Either cards or paper tape may be used for symbolic input to the compiler. External programs that are stored on paper tape, however, must be punched in a format which is slightly different from that used for cards. For this reason, a separate discussion of each of the two alternatives follows.

Machine-Language Procedure Deck

An external program card deck may consist of three parts: a name card, equivalence card (or cards) and the machine-language instruction cards which define the operation of the program.

First: THE NAME CARD

'Name' cards have a two in column 1, followed by the name of the EXTERNAL statement or procedure, a comma, and the type (if necessary).

Second: THE EQUIVALENCE CARDS

'Equivalence' cards have a two in column 1, an identifier, an equal sign, and two digits. These cards are used only when necessary to refer explicitly to other identifiers in the program.

Third: THE INSTRUCTION CARD

'Instruction' cards define the program in machine language, and have the following format:

COLUMNS	CARD ENTRY
1 – 2	60
3	The number of instructions on this card
4 – 10	Any identification, serial numbers, etc., that the user desires
11 – 14	These columns are either blank or contain the relative location of the first instruction on this card
15 – 25	The first instruction
26 – 36	The second instruction
37 – 47	The third instruction
48 – 58	The fourth instruction
59 – 69	The fifth instruction
70 – 80	The sixth instruction

As many of these cards as required are used. *The final card of this program must have as its last instruction the* FINISH *pseudo-operation code*: 4 0000 99 0000.

EXAMPLE:

Suppose we wish to define a procedure to detect an overflow condition. In the symbolic deck, prior to the use of this procedure, we would have the declaration:

```
EXTERNAL PROCEDURE OVERFLOW (; ; L)
```

Following the FINISH declaration of the symbolic program, the following deck would appear:

CARD NO.		ENTRIES	
1	2	OVERFLOW	
2	606	0 0000 00 0000	0 0000 01 9999
			8 0410 40 0002
			0 0000 31 9999
			8 0000 42 0000
			1 0000 30 0000
			4 0000 99 0000
3	2	FINISH	

EXAMPLE:

Suppose that an external procedure is to be defined for the complex multiplication:

$$(A + iB)(C + iD) \rightarrow (X + iY)$$

In the symbolic program, the following declaration would appear:

```
EXTERNAL PROCEDURE CMPMULT(A, B, C, D; X, Y); REAL
  CMPMULT;
```

After the FINISH declaration of the symbolic program would be the following deck:

CARD NO.		ENTRIES	
1	2	CMPMULT, REAL	
2	606	0 0000 01 0000	7 0024 01 9999
			8 0410 40 0017
			8 0000 41 0020
			8 0411 40 0010
			8 0000 10 0024
			8 0000 24 0022
3	606	0 0000 02 0006	8 0000 40 0020
			8 0000 11 0023
			8 0000 24 0021
			8 0000 22 0020
			0 0000 40 9999
			8 0000 10 0024
4	606	0 0000 03 0012	8 0000 24 0021
			8 0000 40 0020
			8 0000 10 0023
			8 0000 24 0022
			8 0000 22 0020
			000000 40 9999
5	604	0 0000 04 0018	8 0000 42 0000
			1 0000 30 0000
			4 0000 00 0005
			4 0000 99 0000

EXAMPLE:

Suppose that an external procedure which will allow keyboard input of a single item of data is to be defined. The declaration

EXTERNAL PROCEDURE KEYIN (; X)

appears in the symbolic program. The card deck which follows the FINISH declaration in the symbolic program would be:

CARD NO.		ENTRIES	
1	2	KEYIN	
2	606	0 0000 01 0000	0 0000 01 9999
			8 0410 40 0004
			0 0007 45 0000
			0 0000 08 0000
			0 0000 40 9999
			8 0000 42 0000
3	602	0 0000 02 0006	1 0000 30 0000
			4 0000 99 0000
4	2	FINISH	

EXAMPLE:

If it is desired to use the keyboard to enter a variable ALPHA, an external statement can be written for this purpose. The external statement is a special case of the external procedure, in which the parameter list is empty.

In the symbolic program the declaration

EXTERNAL STATEMENT KEYIN

would appear. Following the FINISH declaration would be the deck:

CARD NO.		ENTRIES	
1	2	KEYIN	
2	2	ALPHA=33	
3	605	0 0000 00 0000	0 0007 45 0000
			0 0000 08 0000
			6 0000 40 3300
			4 0000 30 0000
			4 0000 99 0000

External Procedures on Paper Tape

External programs are stored on paper tape in 14-word blocks. The tape is constructed in essentially the same manner as that of a machine-language procedure deck; the only exception is that the machine-language instructions are packed in a different format. The paper tape must be prepared in the following sequence:

First: THE NAME BLOCK

The header block assumes the same form as that described under *Machine-Language Procedure Deck*, discussed previously in this appendix. The 22-field of the first word of this block will not be scanned. A sufficient number of blank words must be inserted after the pertinent information to complete the block.

Second: THE EQUIVALENCE BLOCKS

These blocks are used only when it is necessary to refer explicitly to other identifiers in the symbolic program, or to other external or library procedures incorporated in the program. The 22-field of the first word of these blocks will not be scanned. Each equivalence statement must have its own unique block of 14 words.

Third: THE INSTRUCTION BLOCKS

'Instruction' blocks of 14 words define the operation of the external program, and may contain a maximum of 13 machine instructions and pseudo-operation codes. Indicated in the 02 field of the first word of each block is the number of instructions contained in that block. The sign of the first word must be a zero. The final block of every external program must have the FINISH pseudo-operation code 4 *iiii* 99 *iiii* as its last instruction, where *iiii* is irrelevant.

External programs are prepared for paper tape in virtually the same manner as for those punched on cards. The only exception is that the information in columns 1-14 has been deleted in the instruction blocks.

EXAMPLE:

Consider the example on complex multiplication given earlier in this appendix. To prepare this procedure for paper tape, it would now have to be rearranged into the following format:

BLOCK NO.	ENTRIES	REMARKS
1	2 CMPMULT, REAL	
2	0 0000 00 0013	Number of instructions in this block.
	7 0024 01 9999	
	8 0410 40 0017	
	8 0000 41 0020	
	8 0411 40 0010	
	8 0000 10 0024	
	8 0000 24 0022	
	8 0000 40 0020	
	8 0000 11 0023	
	8 0000 24 0021	
	8 0000 22 0020	
	0 0000 40 9999	
	8 0000 10 0024	
	8 0000 24 0021	
3	0 0000 00 0009	Number of instructions in this block.
	8 0000 40 0020	
	8 0000 10 0023	
	8 0000 24 0022	
	8 0000 22 0020	
	0 0000 40 9999	
	8 0000 42 0000	
	1 0000 30 0000	
	4 0000 00 0005	
	4 0000 99 0000	

LIBRARY PROCEDURES

Library procedure decks are prepared in a manner similar to that used for the preparation of external procedures, with the possible exception of the name card. In addition to the header cards described previously in this appendix (see *Description of Name Cards*), another form exists which is applicable only in the case of library procedures.

GENERAL FORM:

$\mathcal{G}$, $\mathfrak{FJ}$, $\mathfrak{GJ}$

where $\mathcal{G}$ and $\mathfrak{FJ}$ have the same meaning as explained in *Description of Name Cards*, and $\mathfrak{GJ}$ is the argument type. Library procedures which act as functions of a single argument may specify the type of this argument. Conversion of this type is performed before the program is entered. If the type of the argument is not specified, no such conversion will take place.

EXAMPLE:

The header card of a procedure in the library reads:

FLOAT, REAL (INTEGER)

If X is FLOATING and the expression FLOAT(X) appears in the symbolic program, then the expression will now be equivalent to FLOAT(FIX(X)), since the argument type of FLOAT ( ) must be INTEGER.

Following the final library deck is again a card with the word FINISH. The method of loading these decks onto the tape is discussed in APPENDIX B, *Compiler Operating Instructions*.

EXAMPLE:

A possible library procedure for the inverse-cosine function is reproduced here as an example. This procedure uses the relation:

$$\arccos x = \arcsin(-x) + \frac{\pi}{2}$$

A reference must be made to the ARCSIN procedure. We shall assume also that cell 0047, relative to the ARCSIN procedure, is available for temporary storage and that cell 0033 contains the constant $\pi/2$.

CARD NO.	ENTRIES		
1	2	ARCCOS, REAL	
2	2	ARCSIN=21	
3	606	0 0000 01 0000	0 0000 01 9999
			6 0000 40 2147
			6 0000 11 2147
			6 0000 44 2100
			6 0000 30 2100
			6 0000 22 2133
4	603	0 0000 02 0006	8 0000 42 0000
			1 0000 30 0000
			4 0000 99 0000

THE ERROR-MESSAGE PROCEDURE

The error-message procedure controls the printing of the following types of error indication on the LINE PRINTER:

RESULT OUT OF RANGE IN $\mathcal{P}$ – $\mathfrak{L}(nnnn)$
RESULT UNDEFINED FOR $\mathcal{P}$ – $\mathfrak{L}(nnnn)$
RESULT ILL-DEFINED FOR $\mathcal{P}$ – $\mathfrak{L}(nnnn)$
ARITHMETIC OVERFLOW – $\mathfrak{L}(nnnn)$

where $\mathcal{P}$ is the label of the procedure which caused the printing. If a MONITOR, TRACE, or DUMP declaration has been given, $\mathfrak{L}(nnnn)$ will also be printed, where

£ is the first five characters of the label of the last labeled statement which has been executed, and *nnnn* is the number of times this statement has been executed.

Any machine-language program may use this procedure to give error indications, provided the following conventions are adhered to:

The name ℘ is in the R register, in alphanumeric form, upon entrance to the error message procedure. (In the case of arithmetic overflow, ℘ is ignored.)

Upon entrance, the exit line is in the B register.

Control is transferred to locations 0000, 0007, 0014, or 0021, relative to the beginning address of the error-message procedure, to cause printing of any of the error indications listed above in their respective order.

EXAMPLE:

To illustrate the use of the error-message procedure, consider the library procedure ARCCOS. The relation

$$\cos^{-1} x = \frac{\pi}{2} - \sin^{-1} x$$

is used for calculation and hence an entry must be made to the ARCSIN routine. Also, since $\cos^{-1} x$ is undefined for $| x | > 1$, an entry must be provided to the error-message procedure.

CARD NO.		ENTRIES		REMARKS
1	2	ARCCOS, REAL		
2	2	ARCSIN=1		
3	2	ERROR=13		
4	606	0 0000 01 0000	0 0000 01 0000	
			8 0010 18 0012	
			8 0000 42 0000	Load B with exit line
			8 0000 41 0013	Load R with name 'ACOS'
			6 0000 34 1311	If $\| x \| > 1$, print undefined error message
			8 0000 40 0015	
5	606	0 0000 02 0006	8 0000 11 0015	
			6 0000 44 0100	Get $\sin^{-1}(-x)$.
			6 0000 30 0101	
			8 0000 22 0014	
			8 0000 42 0000	
			1 0000 30 0000	Exit
6	605	0 0000 03 0012	0 5110 00 0600	1.0
			2 4143 56 6200	'ACOS'
			0 5115 70 7963	$\pi/2$
			0 0000 00 0000	Temporary storage
			4 0000 99 0000	End of procedure

INPUT-OUTPUT PROCEDURES

The compiler system utilizes closed input-output routines for the transmittal of information to and from the computer. For the purpose of clarification, these procedures are divided into two categories, which correspond to the phase in which they are used.

Compilation Phase

Two machine-language procedures that form an integral part of the compiler are employed during compilation time. The function of the first procedure is to read the symbolic source statements; the other furnishes a listing of the source language, together with any error messages.

The compiler program is linked to the INPUTMEDIA procedure by the following pair of instructions:

STP *aaaa*
bbbb BUN *aaaa*

where *aaaa* is the address of the first instruction of the procedure, and *bbbb* is the address of the first word of a 16-word input buffer.

The OUTPUTMEDIA procedure is linked by a pair of instructions:

nncf STP *aaaa*
bbbb BUN *aaaa*

where *nn* is the number of words to be displayed on the output device; *c* specifies the digit to be used for printer control, and is to be interpreted according to the activation phrase list given in CHAPTER VIII; *f* is the digit which specifies the format band; *aaaa* is the address of the first instruction of the procedure; and *bbbb* is the address of the output buffer.

The *f* digit determines the type of output requested by the compiler.

TYPE OF OUTPUT	*f* DIGIT	FORMAT BAND
ALGOL Statements	2	2
Error Messages	6	4
Machine Instructions	0	1
Forward References	4	3
Fix-up (see Chapter X, *Object Program Listing*)	8	5

With the exception of the sign digits, input and output procedures both are written in accordance with the conventions established for external programs, discussed previously in this appendix. The special sign digits given below are used by the BAC-220 generator program to control relocation of the input and output procedures used at compile time.

Sign Digits of Zero through Five

Instructions with signs of zero, one, two, three, four, or five are not altered in any way.

Sign Digits of Six and Seven

An instruction with a sign digit of six or seven has its control field (sL=44) and its address field (sL=04) relocated with respect to the first instruction of the input-output procedure. The sign is set to one if the original sign was seven, and to zero if it was six.

Sign Digit of Eight or Nine

An instruction with a sign digit of eight or nine has its address field (sL=04) relocated with respect to the first instruction of the program. The control field remains unaltered. The sign is set to zero if the original sign was eight, and to one if it was nine.

Each card of an input-output procedure deck will contain one instruction punched in columns 37-47 inclusive and a six in column 1. The remainder of the card may be used for whatever purposes the programmer desires, e.g., remarks or sequence numbers. Directions for integrating these procedures with the compiler program are given in APPENDIX A.

A maximum combined storage of 200 locations is available for special input and output procedures. Neither of these procedures may exceed 150 instructions in length.

EXAMPLE:

The coding of the standard CARDATRON output procedure consists of the following:

RELATIVE LOCATION	CODE	REMARKS
0000	0 0000 NOP 0000	
0001	8 0000 LDB 0000	exit line
0002	8 0412 STB 0012	
0003	1 0000 CAD 9999	address of output buffers
0004	0 0000 SRA 0006	
0005	8 0410 STA 0011	
0006	1 0000 CAD 9998	*nncf*
0007	8 4210 STA 0011	
0008	0 0000 SRA 0008	
0009	0 0000 SUB ONE	integer constant 1
0010	8 0000 ADL 0001	
0011	0 2009 CWR 0000	
0012	0 0000 BUN 0000	

Execute Phase

The input-output media may be altered by replacing the standard CARDATRON REED and RITE procedures in the library, as described in APPENDIX A. The linkage to the REED is located inside the READ procedure, and is identical to that used for input at compilation time. Calls on the RITE procedure are contained in the WRITE, MONITOR, and ERROR procedures, as well as in the symbolic dump routine. The linkage employed is of the form described for OUTPUTMEDIA (see *Compilation Phase* in this appendix). The f digit in the control field (sL=44) of the STP command determines the type of information to be transmitted.

TYPE OF OUTPUT	f DIGIT	FORMAT BAND
Headings, Results and Symbolic Dumps	8	5
Monitoring	4	3
Error Messages	6	4

It may be necessary in certain applications to read and punch cards which have a fixed format. In such cases, external programs for specialized forms of input and output often will be employed in place of the conventional READ and WRITE procedures.

It is necessary in these instances to describe the coding produced by the compiler when an INPUT, OUTPUT, or FORMAT declaration is given.

The prime function of the program produced from either an INPUT or OUTPUT declaration is to form a link between those quantities which have been determined at compilation time (i.e., the addresses representing arrays, variables, expressions, etc.) and those quantities which are known only when the compiled program is executed (i.e., quantities either to be read from, or written on, various input or output media). The programs thus must be produced without reference to the routines which will use them.

When the name of an INPUT or OUTPUT declaration is given to a procedure (either a library procedure such as READ or WRITE, or some EXTERNAL procedure) it is in the program-reference field, and thus an address can be assigned to it. It is to this address that the input or output program will refer to determine an 'exit' address to which the program may transfer control. The input-output program, in turn, leaves its own return address in the B register. The A register is used to transmit the actual data. An OUTPUT declaration will put data to be transmitted to output media in the A register. An INPUT declaration will store data from the A register.

When all the relevant data have been transmitted—the input-output string having been exhausted—the sign of the A register will be loaded with the digit nine, which serves as a termination flag. *Note that no information should be transmitted until an entry has been made to the*

INPUT *or* OUTPUT *declaration.* If these strings were vacuous, the A register would be loaded immediately with the termination flag, but it is necessary to enter the routine to obtain this information.

Some examples should make this clearer. Suppose we have the following INPUT declaration:

```
INPUT DATA (A, I, O(I), B)
```

Let us assume that the variables A, I, O(I), and B have been assigned cells 3701, 3702, 2008 + I, and 3703, respectively, and that the coding generated by the compiler for this INPUT declaration starts at cell 0956.

LOCATION			OPERATION	ADDRESS	REMARKS
0956	0	0000	30	0000	
0957	0	0000	42	0957	first entry
0958	0	0002	20	0956	
0959	0	0000	40	3701	second entry (A)
0960	0	0000	42	0960	
0961	0	0002	20	0956	
0962	0	0000	40	3702	third entry (I)
0963	0	0000	42	0963	
0964	0	0002	20	0956	
0965	0	0000	42	3702	fourth entry (O(I))
0966	1	0000	40	2008	
0967	0	0000	42	0967	
0968	0	0002	20	0956	
0969	0	0000	40	3703	fifth entry (B)
0970	0	0009	43	0000	termination flag
0971	0	0000	30	0956	

The following is a keyboard-input external procedure which could use this INPUT declaration.

The procedure-call in the symbolic program would be:

```
KEYIN (; ; DATA)
```

Thus only one parameter (in rA) is supplied to the external procedure. This parameter is the address of the input string DATA. In this particular case rA would contain 0 0000 01 0956.

CARD NO.		ENTRIES	
1	2	KEYIN	
2	606	0 0000 01 0000	0 0000 01 0000
			8 0410 40 0004
			8 0410 40 0005
			8 0401 26 0005
			0 0000 44 9999
			0 0000 30 9999
3	606	0 0000 02 0006	8 0009 33 0011
			8 0412 40 0005
			0 0001 45 0000
			0 0000 08 0000
			8 0000 30 0005
			8 0000 42 0000
4	602	0 0000 03 0012	1 0000 30 0000
			4 0000 99 0000

As an example of an OUTPUT declaration consider the statement:

```
OUTPUT DATB (A, I, X(I), B + A*2)
```

Assume that the compiler has assigned the cells 3095, 3096, 3097, and 2106 + I to the variables A, I, B, and X(I), respectively, and that the coding for this output statement starts at 1159.

The coding would then appear as:

LOCATION			OPERATION	ADDRESS	REMARKS
1159	0	0000	30	0000	
1160	0	0000	10	3095	first entry (A)
1161	0	0000	42	1161	
1162	0	0002	20	1159	
1163	0	0000	10	3096	second entry (I)
1164	0	0000	42	1164	
1165	0	0002	20	1159	
1166	0	0000	42	3096	third entry (X(I))
1167	1	0000	10	2106	
1168	0	0000	42	1168	
1169	0	0002	20	1159	
1170	0	0000	10	3095	fourth entry (B + A*2)
1171	0	0000	24	3095	
1172	0	0000	22	3097	
1173	0	0000	42	1173	
1174	0	0002	20	1159	
1175	0	0009	43	0000	fifth entry
1176	0	0000	30	1159	termination signal

The following describes an external procedure which uses this declarator, and which will transmit the information to the SUPERVISORY PRINTER as output (in integer format).

The symbolic procedure-call might be, for example:

```
SPO (; ; DATB)
```

Thus only one parameter (in rA) would be supplied to the external procedure. This parameter is the address of the output string DATB and, in this particular case, would be 0 0000 00 1159.

CARD NO.	ENTRIES		
1	2	SPO	
2	606	0 0000 01 0000	0 0000 01 0000
			8 0410 40 0004
			8 0000 42 0004
			8 9999 21 0004
			0 0000 44 0000
			1 0000 30 0000
3	606	0 0000 02 0006	8 0009 33 0010
			8 0000 40 0012
			8 0010 09 0012
			8 0000 30 0004
			8 0000 42 0000
			1 0000 30 0000
4	602	0 0000 03 0012	4 0000 00 0001
			4 0000 99 0000

THE FORMAT DECLARATION

The compiler does not produce a program for a FORMAT declaration. It does a certain amount of preprocessing of the format-data list, and inserts it into the program. This preprocessing consists mainly of breaking the list up into computer words, grouped according to the format data-list elements. The sign of the word determines the type of the associated element.

PHRASE FORMAT	WORD FORMAT
@	
nnn@	
@*www*	0 *nnn* @ *www dd*
nnn@*www*	
@*www.dd*	
nnn@*www.dd*	

where @ is any alphabetic character represented in internal machine code, *nnn* is a three-digit numeric field, *www* is a three-digit numeric field, and *dd* is a two-digit numeric field.

( )	1 000 000 *aaaa*
nnn ( )	1 000 *nnn aaaa*

This word corresponds to a parenthesis pair. *The word appears in the element list at a point corresponding to the right parenthesis.* The digits *nnn* are the repeat digits preceding the corresponding left parenthesis. (A zero implies an indefinite repeat.) The address of the word corresponding to the element following the left parenthesis is *aaaa*.

@@@@@	2@@@@@

This word corresponds to five characters of a format string lying between asterisks.

3@@@@@

This is the termination of a format string. One of the @'s will be an * (internal machine code: 14) showing the exact point of termination of the string. Spaces not within a format string are ignored.

Consider the FORMAT declaration:

```
FORMAT F(5F14.8, *F(X) = *, 2(3A13, I5), W0, (5I10, P) )
```

and assume that the compiler assigns the cell 1894 to the first element in F. The following would then be produced. (The symbol # is used here to indicate a space.)

ADDRESS	ELEMENT	REMARKS
1894	0 005 46 014 08	5F14.8
1895	2 14 46 24 67 04	*F(X)
1896	3 00 33 00 14 00	#=#*
1897	0 003 41 013 00	3A13
1898	0 000 49 005 00	I5
1899	1 000 002 1897	2( )
1900	0 000 66 000 00	W0
1901	0 005 49 010 00	5I10
1902	0 000 57 000 00	P
1903	1 000 000 1901	(5I10)
1904	1 000 000 1894	F( )

Although there is a well-defined interpretation of these elements as far as the WRITE procedure is concerned, the programmer is at liberty to employ these elements in an external program, which will interpret them in any way desired.

APPENDIX G

library procedures

IN ORDER TO MAKE ALTERATIONS to the library procedures listed below, or to incorporate additional procedures in the library, it is necessary to follow the instructions given in APPENDIX B. Maintaining the library in this manner requires that the user be in possession of both the BAC-220 Generator Tape and the appropriate library-procedure decks. The preparation of the latter is discussed in APPENDIX F.

A description of these library procedures is given in the following pages, preceded by a specimen page showing the format of these descriptions.

The compiler tape contains the following library of standard procedures:

LIBRARY PROCEDURE NAME	DESCRIPTION
FLOAT	Converts an integer to a floating-point number.
FIX	Truncates a floating-point number to an integer.
READ	Input
WRITE	Output
ERROR	Library error-message procedure (see APPENDIX F).
SQRT	Square-root function
EXP	e^x
LOG	ln x
FX*FX, FL*FX, FL*FL, FX*FL	Power routines
SIN	sin x
COS	cos x
TAN	tan x
ARCSIN	$\sin^{-1} x$
ARCCOS	$\cos^{-1} x$
ARCTAN	$\tan^{-1} x$
ROMXX	$(1 - x^2)^{1/2}$
ENTIRE	Greatest integer $[x]$
LABEL	Library procedure employed for minimum monitoring (see CHAPTER X).
MONIT	Library monitor procedure (see CHAPTER X).
TRACE	Controls symbolic memory dump action (see CHAPTER X).
SINH	sinh x
COSH	cosh x
TANH	tanh x
REED, RITE	Special input-output procedures (see APPENDIX F).

(NAME)

FORM	The generic form used by the programmer in his source program, the italicized letters indicating the input arguments for the procedures.
ARGUMENT	Specifies the type of argument required for the procedure.
RESULT	Defines the type of result produced by the procedure.
DESCRIPTION	Outlines the operation carried out by the procedure.
ACCURACY	Self-explanatory
ERROR MESSAGE	Lists all error messages which will be printed out for the given procedure. This is shown as the error message itself, followed by two blank columns, a hyphen, the first ten characters of £ (the label of the last labeled statement which has been executed), and (*nnnn*), the number of times this statement has been executed. If no MONITOR statement precedes the program, or no labeled statement has been executed, blank columns will appear in place of £ and (*nnnn*).
EXTERNAL USE	Explains conditions to be met in order to use these procedures with external machine-language procedures. The name given here is to be used on equivalence cards in external programs. (See APPENDIX F.)
REMARKS	Self-explanatory

FLOAT

FORM	FLOAT (x)
ARGUMENT	x is integral.
RESULT	FLOAT (x) is floating-point.
DESCRIPTION	Converts the argument x into its corresponding floating-point form, as defined under ACCURACY.
ACCURACY	Exact for $\mid x \mid < 10^8$, otherwise the result is truncated to eight significant digits.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. On entry to the procedure, the exit from the subroutine must be stored in the 04 field of the first line of the procedure. Result of the procedure FLOAT (x) is in rA. Use FLOAT on equivalence cards.
REMARKS	

FIX

FORM	FIX (x)
ARGUMENT	x is floating-point.
RESULT	FIX (x) is integral.
DESCRIPTION	Truncates the argument x into its corresponding integral form. Any fractional part is lost.
ACCURACY	
ERROR MESSAGE	RESULT OUT OF RANGE IN FIX $-£(nnnn)$ This print-out will result if $\mid x \mid \geq 10^{10}$
EXTERNAL USE	x is in rA. Use FIX on equivalence cards.
REMARKS	

WRITE

FORMS

First form:
WRITE (; OUTDEC, FRMTDEC).

Second form:
WRITE (; ; FRMTDEC)

ARGUMENTS

OUTDEC is an identifier declared to be the label of an output-data list. FRMTDEC is an identifier declared to be the label of a format-data list.

RESULT

DESCRIPTIONS

First form:
Print or punch the output-data list OUTDEC as output on the LINE PRINTER, the CARD PUNCH, or the SUPERVISORY PRINTER according to the format FRMTDEC.

Second form:
Print or punch messages as output on the LINE PRINTER, the CARD PUNCH, or the SUPERVISORY PRINTER as given by the format FRMTDEC.

ACCURACY

ERROR MESSAGE

None

CALLING SEQUENCE

First form:

```
          CAD    (Address of OUTDEC)
          DLB    WRITE sL = 44
                        nn = 00
          STA    -0
          CAD    (Address of FRMTDEC)
0  0000   STP    WRITE
0  0100   BUN    WRITE
```

Second form:

```
          CAD    (Address of FRMTDEC)
          STP    WRITE
0  0000   BUN    WRITE
```

REMARKS

For further details, see CHAPTER VIII and APPENDIX F.

READ

FORMS

First form:
READ (; ; INDEC)
Second form:
READ (; S ; INDEC)

ARGUMENTS

INDEC is an identifier declared to be the label of an input-data list.

RESULT

S is a Boolean variable.

DEFINITIONS

First form:
Read in the input-data list INDEC from the CARD READER.

Second form:
Same as the first form, but in addition, if the word SENTINEL is encountered (other than in an alphanumeric entry), terminate the input process and set S to one. If not, set S to zero.

ACCURACY

ERROR MESSAGE

None

CALLING SEQUENCE

First form:

```
          CAD    (Address of INDEC)
          STP    READ
0  0000   BUN    READ
```

Second form:

```
          CAD    (Address of S)
          DLB    READ, sL = 44
                       nn = 00
          STA    -0
          CAD    (Address of INDEC)
          STP    READ
0  0100   BUN    READ
```

REMARKS

For further details, see CHAPTER VIII and APPENDIX F.

SQRT

FORM	SQRT (x)
ARGUMENT	x is floating-point.
DESCRIPTION	SQRT (x) is the square root of x.
ACCURACY	The maximum error is 2 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR SQRT -£($nnnn$) This printout will result if $\|x\| < 0$.
EXTERNAL USE	x is in rA. Use SQRT on equivalence cards.
REMARKS	

EXP

FORM	EXP (x)
ARGUMENT	x is floating-point.
RESULT	EXP (x) is floating-point.
DESCRIPTION	EXP (x) computes the exponential function e^x.
ACCURACY	Let ϵ be the error in the eighth significant digit, then for $\|x\| < 100$, $\epsilon \leq 3$; for $100 \leq \|x\| < 112.82666$, $\epsilon \leq 6$.
ERROR MESSAGE	RESULT OUT OF RANGE IN EXP -£(*nnnn*) This printout will result if $\|x\| \geq 112.82666$.
EXTERNAL USE	x is in rA. Use EXP on equivalence cards.
REMARKS	

LOG

FORM	LOG (x)
ARGUMENT	x is floating-point.
RESULT	LOG (x) is floating-point.
DESCRIPTION	LOG (x) is the natural logarithm, ln x.
ACCURACY	The maximum error is 9 in the eighth significant digit
ERROR MESSAGE	RESULT UNDEFINED FOR LOG -£(*nnnn*) This printout will result if $x \leq 0$.
EXTERNAL USE	x is in rA. Use LOG on equivalence cards.
REMARKS	

POWER
ROUTINE

FLFL

FORM — *A*B*

ARGUMENTS — *A* is floating-point; *B* is floating-point.

RESULT — The result, A^B, is floating-point.

DESCRIPTION

ACCURACY — The error is, in general, less than 8 in the eighth significant digit. However, since the error is a function of the magnitude of A^B, the maximum error is 3 in the sixth significant digit.

ERROR MESSAGES — RESULT OUT OF RANGE IN FLFL -£(*nnnn*)
This printout will result if $|A^B| > 0.99999999 \times 10^{49}$.
RESULT UNDEFINED FOR FLFL -£(*nnnn*)
This printout will result if $A = 0$ and $B \leq 0$, or if $A < 0$ and B is non-integral.

EXTERNAL USE — *A* is in rA; *B* is in rR. Use FL*FL on equivalence cards.

REMARKS — This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.*
Note that the error messages for this routine are identical to those which can occur in the power routine FXFL.

POWER
ROUTINE

FLFX

FORM	$A*B$
ARGUMENTS	A is floating-point; B is integral.
RESULT	The result, A^B, is floating-point.
DESCRIPTION	
ACCURACY	The result, A^B, has a maximum error of $\log_2 B$ in the eighth significant digit.
ERROR MESSAGES	RESULT OUT OF RANGE IN FLFX -£(*nnnn*) This printout will result if $\|A^B\| > 0.99999999 \times 10^{49}$. RESULT UNDEFINED FOR FLFX -£(*nnnn*) This printout will result if $A = 0$ and $B \leq 0$.
EXTERNAL USE	A is in rA; B is in rR. Use FL*FX on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.*

POWER
ROUTINE

FXFL

FORM	$A*B$
ARGUMENTS	A is integral; B is floating-point.
RESULT	The result, A^B, is floating-point.
DESCRIPTION	
ACCURACY	The error is, in general, less than 8 in the eighth significant digit. However, since the error is a function of the magnitude of A^B, the maximum error is 3 in the sixth significant digit.
ERROR MESSAGES	RESULT OUT OF RANGE IN FLFL -£(*nnnn*) This printout will result if $\|A^B\| > 0.99999999 \times 10^{49}$. RESULT UNDEFINED FOR FLFL -£(*nnnn*) This printout will result if $A = 0$ and $B \leq 0$, or if $A < 0$ and B is non-integral.
EXTERNAL USE	A is in rA; B is in rR. Use FX*FL on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.* Note that the name of the power routine FLFL occurs incorrectly in the error messages for this routine.

POWER
ROUTINE

FXFX

FORM	$A*B$
ARGUMENTS	A and B are integers.
RESULT	The result, A^B, is an integer. If $A \neq 0$ or 1, and $B < 0$, then the result is the integer 0.
DESCRIPTION	
ACCURACY	The result is exact.
ERROR MESSAGES	RESULT OUT OF RANGE IN FXFX -£(*nnnn*) This printout will result if $\|A^B\| \geq 10^{10}$. RESULT UNDEFINED FOR FXFX -£(*nnnn*) This printout will result if $A = 0$ and $B \leq 0$.
EXTERNAL USE	A is in rA; B is in rR. Use FX*FX on equivalence cards.
REMARKS	This routine is automatically included in the compiled program as required. *It is never called explicitly as a procedure.*

SIN

FORM	SIN (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	SIN (x) is floating-point.
DESCRIPTION	SIN (x) computes the sine function.
ACCURACY	The maximum error is 2 in the eighth significant digit.
ERROR MESSAGE	RESULT ILL-DEFINED FOR SIN -£(*nnnn*) This printout will result if $\mid x \mid \geq 10^7$ radians.
EXTERNAL USE	x is in rA. Use SIN on equivalence cards.
REMARKS	

COS

FORM	COS (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	COS (x) is floating-point.
DESCRIPTION	COS (x) computes the cosine function.
ACCURACY	The maximum error is 24 in the last two digits.
ERROR MESSAGE	RESULT ILL-DEFINED FOR COS -£(*nnnn*) This printout will result if $\mid (x + \pi/2) \mid \geq 10^7$ radians.
EXTERNAL USE	x is in rA. Use COS on equivalence cards.
REMARKS	

TAN

FORM	TAN (x)
ARGUMENT	x is in radians, and is floating-point.
RESULT	TAN (x) is floating-point.
DESCRIPTION	TAN (x) computes the tangent function.
ACCURACY	The error is less than 21 in the last two significant digits, except near $k(\pi/2)$ where k is an odd integer.
ERROR MESSAGE	RESULT UNDEFINED FOR TAN -£($nnnn$) This printout will result if $\cos x = 0$. RESULT ILL-DEFINED FOR TAN -£($nnnn$) This printout will result if $\mid x \mid \geq 10^7$ radians.
EXTERNAL USE	x is in rA. Use TAN on equivalence cards.
REMARKS	

ARCSIN

FORM	ARCSIN (x)
ARGUMENT	x is floating-point.
RESULT	ARCSIN (x) is in radians, and is floating-point.
DESCRIPTION	ARCSIN (x) computes the inverse-sine function. The principal range is $[-\pi/2, \pi/2]$.
ACCURACY	The maximum error is 7 in the eighth significant sigit.
ERROR MESSAGE	RESULT UNDEFINED FOR ASIN -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ARCSIN on equivalence cards.
REMARKS	

ARCCOS

FORM	ARCCOS (x)
ARGUMENT	x is floating-point.
RESULT	ARCCOS (x) is in radians, and is floating-point.
DESCRIPTION	ARCCOS (x) computes the inverse-cosine function. The principal range is [0, π].
ACCURACY	Over most of interval [0, 1] the maximum error will be 9 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR ACOS -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ARCCOS on equivalence cards.
REMARKS	

ARCTAN

FORM	ARCTAN (x)
ARGUMENT	x is floating-point.
RESULT	ARCTAN (x) is in radians, and is floating-point.
DESCRIPTION	ARCTAN (x) computes the inverse-tangent function. The principal range is $[-\pi/2, \pi/2]$.
ACCURACY	The maximum error is 4 in the eighth significant digit.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. Use ARCTAN on equivalence cards.
REMARKS	

SINH

FORM	SINH (x)
ARGUMENT	x is floating-point.
RESULT	SINH (x) is floating-point.
DESCRIPTION	SINH (x) computes the hyperbolic-sine function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then if $\mid x \mid < 100$, $\epsilon \leq 7$; if $100 \leq \mid x \mid \leq 112.82666$, $\epsilon \leq 13$.
ERROR MESSAGE	RESULT OUT OF RANGE IN SINH -£(*nnnn*) This printout will result if $\mid x \mid \geq 112.82666$.
EXTERNAL USE	x is in rA. Use SINH on equivalence cards.
REMARKS	

COSH

FORM	COSH (x)
ARGUMENT	x is floating-point.
RESULT	COSH (x) is floating-point.
DESCRIPTION	COSH (x) computes the hyperbolic-cosine function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then for $\mid x \mid < 100$, $\epsilon \leq 7$. for $100 \leq \mid x \mid \leq 112.82666$, $\epsilon \leq 13$.
ERROR MESSAGE	RESULT OUT OF RANGE IN COSH -£(*nnnn*) This printout will result if $\mid x \mid \geq 112.82666$.
EXTERNAL USE	x is in rA. Use COSH on equivalence cards.
REMARKS	

TANH

FORM	TANH (x)
ARGUMENT	x is floating-point.
RESULT	TANH (x) is floating-point.
DESCRIPTION	TANH (x) calculates the hyperbolic-tangent function.
ACCURACY	Let ϵ be the error in the eighth significant digit; then for $\mid x \mid < 100$, $\epsilon \leq 10$; for $\mid x \mid \geq 100$, the result is exact.
ERROR MESSAGE	None
EXTERNAL USE	x is in rA. Use TANH on equivalence cards.
REMARKS	

ROMXX

FORM	ROMXX (x)
ARGUMENT	x is floating-point.
RESULT	ROMXX (x) is floating-point.
DESCRIPTION	ROMXX computes the function $(1 - x^2)^{1/2}$.
ACCURACY	The maximum error is 2 in the eighth significant digit.
ERROR MESSAGE	RESULT UNDEFINED FOR ROMXX -£(*nnnn*) This printout will result if $\mid x \mid > 1$.
EXTERNAL USE	x is in rA. Use ROMXX on equivalence cards.
REMARKS	In order to obtain accuracy for x near unity, double-precision arithmetic is used. Note that $(x^2 - 1)^{1/2} = x$. ROMXX $(1.0/x)$.

ENTIRE

FORM	ENTIRE (x)
ARGUMENT	x is floating-point.
RESULT	ENTIRE (x) is floating-point.
DESCRIPTION	ENTIRE (x) computes the function normally denoted by $[\, x \,]$; it is defined to be the largest integer not greater than x.
ACCURACY	
ERROR MESSAGES	None
EXTERNAL USE	x is in rA. Use ENTIRE on equivalence cards.
REMARKS	

index

[References to definitions of terms are in italics.]

ABS (intrinsic function), 7-3
Activation phrases (for output), *8-5*
Addition (+), arithmetic operator, 3-1
ALGOL transliteration, APPENDIX E
Alphanumeric input, rules for, 8-3
Alphanumeric output, editing for, 8-4
Alternative (EITHER IF) statement, *6-3*
AND (logical operator), 3-3
Arguments,
  of functions, *2-3*
  of procedures, 7-6
Arithmetic expressions, 3-1
Arithmetic operators, 3-1
ARRAY declaration, 5-2
  construction of, 5-2
Arrays, method of filling, 5-2
Assignment statement, *4-1*
  arithmetic, 4-1
  Boolean, 4-1
  generalized, 4-2
Asterisk, 2-1
  on data cards, 8-3
  for exponentiation, 3-1
  in format strings, 8-4, F-11
  printing of, 8-5
Asterisks,
  for floating-point scale factor, 2-3

BAC-220, non-standard versions, A-5
  definition of, *A-5*
  generation of, A-5
    HIGH-SPEED PRINTER version, A-6
    paper-tape version, A-7
BAC-220, standard version, A-2
  definition of, *A-2*
  generation of, A-2
Basic symbols,
  syntactical description of, APPENDIX D
  transliteration rules for, APPENDIX E
BEGIN (statement parenthesis), 4-2
Blanks, insertion in output line, 8-6
Boolean constants, *2-3*
BOOLEAN declaration (of type), 5-1
Boolean expressions, *3-3*
  construction of, 3-3
Boolean quantities, 2-2

C-digit, for printer control, 8-5
Callout decks, B-1
Card format,
  data cards, 8-2
  equivalence cards, F-3
  external procedure, APPENDIX F
  SENTINEL cards, 8-3
  source deck, B-1
Cell-count message, B-2
Character set used by compiler, 2-1
Clauses, 6-2
Commas,
  in subscript lists, 2-2, 5-3
  in type lists, 5-1
COMMENT declaration, 5-3
COMPILED OBJECT PROGRAM CALLOUT deck, B-1
COMPILED OBJECT PROGRAM DUMP CALLOUT deck, B-1
COMPILED OBJECT PROGRAM LOADER deck, B-3
COMPILED OBJECT PROGRAM LOADER BOOTSTRAP deck, B-1
Compiled program listing, 10-6
COMPILER CALLOUT deck, B-1
Compiler language,
  ALGOL transliteration of, APPENDIX E
  characters used in, 2-1
  syntax of, APPENDIX D
Compiler operating instructions, APPENDIX B
Compiler-system tape, use of, B-1
Compiler version statements, A-1, A-2
Compound statements, *4-2*
Conditional control, 6-2
Constants (Boolean, floating-point, and integer), 2-2
Control statements,
  for iteration, 6-4
  for suspension of computation, 6-2
  for transfer (of control), 6-1, 7-1
Control switch, program (*see* PROGRAM CONTROL SWITCH)
Corrections statement, A-1, A-3

Data cards, preparation of, 8-2
Data tape (paper), preparation of, 8-3
Debugging (*see* Diagnostic Aids)
Decimal point,
  card-equipment symbol for multiplication, E-1
Declaration,
  ARRAY, 5-2
    construction of, 5-2
  DUMP, 10-4
  EXTERNAL program, 7-7
  FINISH, 5-3
  FUNCTION, 7-2
  MONITOR, 10-4
  of type, *5-1*
    construction of, 5-1
    by default, *5-2*
    restrictions upon, 7-2
  PROCEDURE, 7-4
    examples of, 7-5
  SEGMENT, 9-1
  SUBROUTINE, 7-1
  TRACE, 10-5
Diagnostic aids, CHAPTER X
  error messages,
    during compilation, 10-1, B-2
    during execution, from library procedures, 10-6
    during generation of compiler, A-9
    spurious, 10-1
  memory dumps,
    machine-language program, B-3
    symbolic, 10-4, 10-5

Index (continued)

Diagnostic Aids (continued)
  Object program,
    dumping of, 10-4, B-3
    listing of, 10-6
    monitoring of, 10-4, 10-5
    tracing of, 10-4
Diagnostic declarations,
  DUMP declaration, 10-4
  MONITOR declaration, 10-4
  TRACE declaration, 10-5
Division (/), arithmetic operator, 3-1
Dollar sign, print for semicolon, 2-1
Dummy statement, labeled, use of, 4-3
DUMP declaration, 10-4
Dump list, *10-5*
Dumps,
  machine-language program, B-3
    on paper tape, B-3
    on punched cards, B-3
  symbolic storage, 10-4
    manual, 10-5
    program-controlled, 10-4, 10-5

Editing elements (for output), 8-4
EITHER IF (alternative) statement, *6-3*
END (statement parenthesis), 4-2
ENTER statement, 7-2
EQIV (logical operator), 3-3
EQL (relational operator), 3-2
Equality sign, as symbol for substitution, 4-1
Equivalence cards, F-3
Error messages,
  during compilation, 10-1, B-2
  during execution, from library procedures, 10-6
  during generation of compiler, A-9
  spurious, 10-1
Error message procedure, F-7
Evaluated functions, *2-3*, 7-2
Exponentiation (*), arithmetic operation, 3-1
Expressions, CHAPTER III
  arithmetic, 3-1
  Boolean, 3-3
  mixed, 3-2
  syntactical description of, APPENDIX D
  transliteration rules for, APPENDIX E
EXTERNAL PROCEDURE
  declaration, 7-8, F-2, F-6
External procedures, *7-4*, APPENDIX F
External programs, preparation of, APPENDIX F
EXTERNAL STATEMENT
  declaration, 4-3, 7-7, F-6

FINISH declaration, 5-3
  operation of, B-2
  use in external programs, F-5
FINISH statement, A-2, A-5, A-6
Fixed-point (integer) quantities, 2-2
FLOATING declaration, 5-1
  by default, 5-2
Floating-point constants, *2-2*
Floating-point quantities, *2-2*
FOR statement, 6-5
Format data-list, *8-4*, F-11
Format data-list elements, *8-4*, F-11
FORMAT declaration, 8-4, F-11
Format string, *8-4*, 8-5, F-11
Function call, *7-2*
FUNCTION declaration, 7-2
Functions, 7-2
  arguments of, *2-3*
  declared, 7-2
  defined by procedures, 7-7
  evaluated, *2-3*, 7-2
  intrinsic, 7-3
    table of, 7-3
  used as arguments, 7-7

GENERATOR CALLOUT deck, A-1, A-2, B-1
Generator input deck,
  composition of, A-1
    compiler version statements, A-1
    corrections statement, A-1
    FINISH statement, A-2
    GENERATOR CALLOUT deck, A-1
    input-output facilities statements, A-2
    miscellaneous option statements, A-2
    system environment statements, A-1
Generator input statements, A-1
Generator operating instructions, APPENDIX A
GEQ (relational operator), 3-2
GO statement, 6-1
GO TO statement, 6-1
GTR (relational operator), 3-2
Header (name) card preceding external program, F-1
HIGH-SPEED PRINTER version of BAC-220, A-6

Identifiers, *2-1*
  reserved, list of, APPENDIX C
IF statement, 6-2
  nested, 6-4
IMPL (logical operator), 3-3
Induction variable, *6-5*
Input cards, preparation of, 8-2
Input data-list elements, *8-1*
Input-data lists, *8-1*
INPUT declaration, 8-1
Input label, *8-1*
Input-list element, *8-1*
Input-output combinations, APPENDIX A
  table of, A-2
Input-output facilities statements, A-2, A-6
Input-output procedures, machine-language, F-8
Input-output routines, non-standard, A-8
Input-output techniques, CHAPTER VIII
Input procedures, symbolic, 8-2
Input tape (paper), preparation of, 8-3
Integers, arithmetic, combinations of, 3-2
INTEGER declaration, 5-1
Integer quantities, *2-2*
Intrinsic functions, *7-3*
  table of, 7-3
Insufficient symbol storage space, corrective measures for, 10-1
Iterated variables, *8-1*
Iteration, control of, 6-1

Labels,
  for input, *8-1*
  for output, *8-3*
  for statements, *4-2*
Leading zeros, 2-2, A-1
LEQ (relational operator), 3-2
Library procedures, *7-4*, F-7
  description of, APPENDIX G
  error messages from, 10-6
  list of, G-1
  listing of, 10-7
Linkage to machine-language procedures, F-1
List of parameters, in procedure declaration, *7-4*
Listing,
  of external programs, 10-7
  of library procedures, 10-7
  of object program, 10-6
  of symbolic program, 10-1
Logical (Boolean) operators, 3-3
  precedence of, 3-3
LSS (relational operator), 3-2

Machine-language procedure deck, F-5
Machine-language procedures, 7-4, 7-8
  construction of, APPENDIX F
Machine-language program dumps,
  on paper tape, B-3
  on punched cards, B-3
Machine-language programs,
  construction of, APPENDIX F
Magnetic tape, use of,
  during compilation, B-2
  in machine-language programs, F-3
  in reloading compiled program, B-3
  in running generator program, A-1
MAX (intrinsic function), 7-3
Memory dump of,
  machine-language programs,
    on paper tape, B-3
    on punched cards, B-3
  symbolic,
    manual, 10-5
    program-controlled, 10-4, 10-5

Index (continued)

Memory space, conservation of, 10-1
Metalinguistic symbols, 2-1
MIN (intrinsic function), 7-3
Minimum monitoring, *10-5*
Miscellaneous option statements, A-2, A-4
Missing address in object program listing, 10-6
Mixed (floating and fixed) arithmetic, 3-2
MOD (intrinsic function), 7-3
MONITOR declaration, 10-4
Monitoring (of object program), 10-4
  of control sequence, 10-4
  minimum, *10-5*
  of variables, 10-4
Monitor list, *10-4*
Multiplication (.), arithmetic operator, 3-1
Multiplication sign,
  decimal point equivalent of, 3-1
  omission of, 3-1

Name (header) card preceding external program, F-4
NEQ (relational operator), 3-2
Nested IF statements, 6-4
Non-standard version of BAC-220,
  definition of, *A-5*
  generation of, A-5
    HIGH-SPEED PRINTER version, A-6
    paper-tape version, A-7
Non-standard input-output routines A-8
NOT (logical operator), 3-3

Object program
  cell count message, B-2
  dump of,
    on paper tape, B-3
    on punched cards, B-3
  listing of, 10-6
  monitoring of, 10-4
  reloading of, B-3
Operators, CHAPTER III
  arithmetic, 3-1
  logical, 3-3
  relational, 3-2
  rules of precedence for sequencing, 3-1, 3-3
Operating instructions for BAC-220, APPENDIX B
Operating instructions for BAC-220 generator, APPENDIX A
OR (logical operator), 3-3
OTHERWISE, used in alternative statements, 6-3
Output-data lists, *8-3*
OUTPUT declaration, 8-3
Output labels, *8-3*
Output procedures, symbolic, 8-4
OVERLAY statement, 9-2
Overlay techniques, 9-1

Page-eject control, 8-6
Paper tape,
  dumping compiled program on, B-3
  external procedures on, F-6
  format of, 8-3, F-7
  preparation of data on, 8-3
  preparation of source programs on, B-1
  reloading object program from, B-3
  SENTINEL on, 8-3
Paper-tape version of BAC-220, A-7
Parameters,
  lists of, *7-4*
  of name, *7-5*
  of value, *7-5*
Parentheses,
  optional use of, 5-1, 8-1, 8-4
  in parameter lists, 7-4
  in place of BEGIN, 4-2
  in place of END, 4-2
  printing equivalents of, 2-1
  in repeat lists, 8-6
  use to indicate precedence, 3-1, 3-3
PCS (intrinsic function), 7-3
Phrases,
  activation, *8-5*
  repeat, 8-6
Power routines, APPENDIX G
Precedence rules,
  arithmetic, 3-1
  Boolean, 3-3
Prefixes, use of, 5-1
Preparation of data,
  on paper tape, 8-3
  on punched cards, 8-2
Printed characters, 2-1
Procedure-assignment statement, *7-7*
Procedure-call statement, *7-6*
  examples, of 7-6
PROCEDURE declaration, 7-4
  examples of, 7-5
Procedures,
  arguments of, 7-6
  availability to compiled programs, 7-4
  declaration of, 7-4
  declarations within, 7-4
  external, *7-4*, APPENDIX F
  functions defined by, 7-7
  input-output, 8-2, 8-4, F-8
  library, *7-4* APPENDIX G
  linkage to, F-1
  machine-language, 7-4
  parameter lists in, *7-4*
  parameters of, F-1
PROGRAM CONTROL SWITCH
  during compile phase,
    number 1 — 10-7, B-2
    number 2 — 10-6, B-2
    number 3 — 10-7, B-2
    number 4 — 10-1, B-2
  during execute phase,
    number 0 — 10-4
    number 7 — 10-6
    number 8 — 10-6
    number 9 — 10-5
  during generation,
    number 1 — A-1
  interrogation of, 7-3
Program logic, verification of, CHAPTER X
Programs, machine-language, APPENDIX F
Punched cards,
  equivalence, F-3
  external procedures on, APPENDIX F
  format of, 8-2, B-1
  loading compiler routines from, B-1
  preparation of data for, 8-2
  punching object program on, B-3
  in running generator program, A-1
Punching out object program,
  on paper tape, B-3
  on punched cards, B-3

Quantities (Boolean, floating-point, and integer), *2-2*

READ procedure, 8-2
REAL declaration, 5-1
REED procedure, *8-2*
Relational operator, 3-2
Relation, *3-2*
  enclosed within parentheses, 3-3
  form of condition, *3-2*
  use in control statements, 3-2
  use in statement sequencing, 3-2
Relocation base, *F-2*
Relocation conventions, F-2
Relocation digits,
  sign digit of zero — F-2, F-8
  sign digit of one — F-2, F-8
  sign digit of two — F-2, F-8
  sign digit of three — F-2, F-8
  sign digit of four — F-2, F-8
  sign digit of five — F-3, F-8
  sign digit of six — F-3, F-9
  sign digit of seven — F-3, F-9
  sign digit of eight — F-3, F-9
  sign digit of nine — F-3, F-9
Repeat phrases, 8-6
Reserved words, list of, APPENDIX C
RETURN statement, 7-1
RITE procedure, *8-4*

Sample programs, CHAPTER XI
Scale factors in floating-point constants, 2-3
SEGMENT declaration, 9-1
Segmentation, 9-1

Index (continued)

Semicolon,
on data cards, 8-3
in FINISH declaration, 5-3
in grammar of statements, 4-2
printing equivalent of, 2-1
in READ procedure, 8-2
in WRITE procedure, 8-4
Sentinel block, *8-3*
Sentinel card, 8-2
format, 8-2
in READ procedure, 8-2
Sentinel declaration,
on paper tape, 8-3
on punched cards, 8-3
SIGN (intrinsic function), 7-3
Simple variables, 2-2
Source program, preparation of,
by means of paper tape, B-1
by means of punched cards, B-1
Spaces, 2-1, 3-2, F-11
Spacing control, for printing, 8-6
Special input-output routines, A-8
Standard version of BAC-220, A-2
generation of, A-2
input-output requirements for, A-2
Statements, *4-1*
alternative, *6-3*
assignment, *4-1*
arithmetic, 4-1
Boolean, 4-1
generalized, 4-2
Statements (continued)
compound, *4-2*
control, 4-1, CHAPTER VI
generator input, A-1
grammar of, 4-2
monitoring of, 10-4
procedure-assignment, *7-7*
procedure-call, *7-6*
syntactical description of, APPENDIX D
transliteration rules for, APPENDIX E
STOP statement, 6-2
Stopping of compiler, unintentional, 10-1
Subroutine, *7-1*
Subroutine call (*see* ENTER statement)
SUBROUTINE declaration, 7-1
Subroutine exit (*see* RETURN statement)
Subscripted variables, 2-2, 5-2
Subtraction (—), arithmetic operator, 3-1
Suspension of computation, 6-2
SWITCH statement, 6-1
Switch, program control (*see* PROGRAM CONTROL SWITCH)
Symbolic program (*see* Source Program)
Syntactical check of source program, CHAPTER X
Syntax, errors in, 10-1
Syntax of compiler language, APPENDIX D
System environment statements, A-1, A-3

TO (*see* GO TO)
TRACE declaration, 10-5
Trace list, *10-5*
Transfer of control, conditional,
alternative statement, *6-3*
FOR statement, 6-5
IF statement, 6-2
nested, 6-4
SWITCH statement, 6-1
UNTIL statement, 6-4
Transfer of control, unconditional,
ENTER statement, 7-2
GO (or GO TO) statement, 6-1
RETURN statement, 7-1
Type,
in arithmetic assignment statements, 4-1
of arithmetic expressions, 3-2
in Boolean assignment statements, 4-1
in generalized assignment statements, 4-2
Type declarations, 5-1
by default, 5-2
restrictions upon 7-2
Type list, *5-1*

UNTIL statement, 6-4

Variable,
induction, *6-5*
iterated, *8-1*
Variables,
simple, 2-2
subscripted, 2-2, 5-2
use in iterations, 6-5

WRITE procedure, 8-4

Burroughs Corporation DETROIT 32, MICHIGAN

offices in principal cities

in CANADA: BURROUGHS BUSINESS MACHINES LTD.
TORONTO, ONTARIO